Introduction to Interscript

19 September 1985

Table of contents

1

Document interchange overview

1.1 The problem

Before the advent of word processors people had no problem interchanging documents. They simply distributed typed sheets of paper. If someone wanted to "pick up" words in an existing document, they retyped them. But now that documents exist in electronic form, people try to avoid retyping anything. They want to exchange electronic documents and "pick up" sections of such documents without actually handling paper at all.

Such exchanges can be smooth when the document is accepted by the same hardware and software that created it. Unfortunately even small office environments have different document creation devices that usually cannot "speak" to each other.

The desire to interchange documents electronically among dissimilar machines is so critical several companies have found a market for devices that convert documents from a few electronic formats into a few others. That's the function of the "black box" shown in Figure 1.1. A document "arrives" over a communication link or on magnetic media.

This is, of course, an *ad hoc* solution. The "black box" doesn't support every format and is always in danger of becoming obsolete as manufacturers change their electronic formats to provide increased functionality. And since "black boxes" are needed wherever documents are created, they represent a significant incremental expense for what seems like such a simple capability.

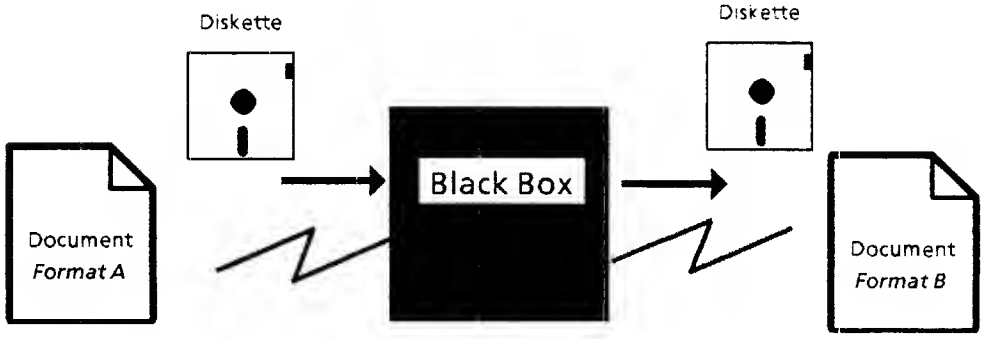

Figure 1.1: Interchanging dissimilar document formats.

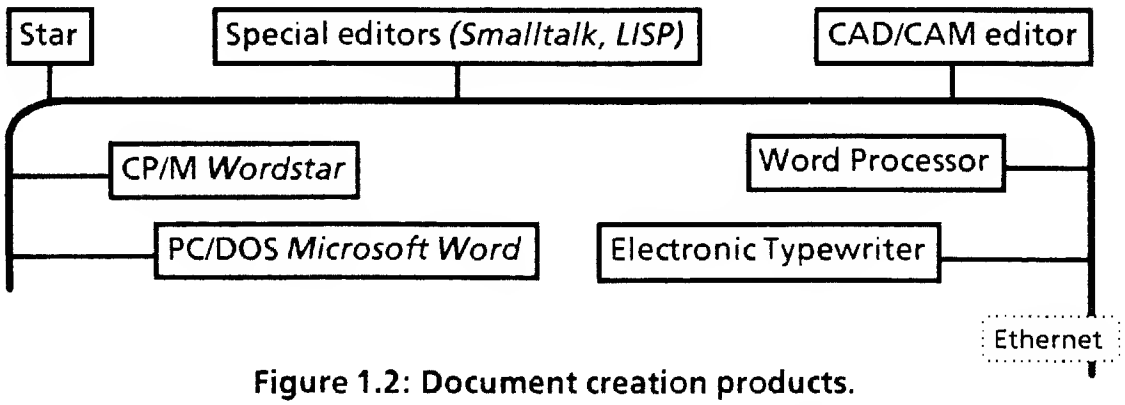

Figure 1.2: Document creation products.

The people within offices have traditionally installed diverse products from different manufacturers. Figure 1.2 depicts seven different hardware/software products offered by a variety of vendors. All attach to a local area network and can create, store, and retrieve documents in electronic form. That does not mean, however, a particular device can edit or display what it retrieves if the document was created by another product.

Why don't manufacturers provide routines that convert from their product's internal electronic format to the formats of others? Because it is too expensive for even a small number of products. Each arrow in Figure 1.3 represents one conversion routine, from the format at the base of the arrow to the format at the head of the arrow. There are 42 arrows in this figure. Adding just one more product increases the number of conversion routines (arrows) to 56. And this is just to provide document interchange among a very small number of products.

Even if an office had access to all 42 conversion routines, it wouldn't be good enough. That's because converting from one format to another loses information. The conversion of a Star document, for example, to a word processor format would preserve text characters (assuming the characters are recognizable by the word processor), but it would not preserve font information such as character height and italics, and certainly not graphics

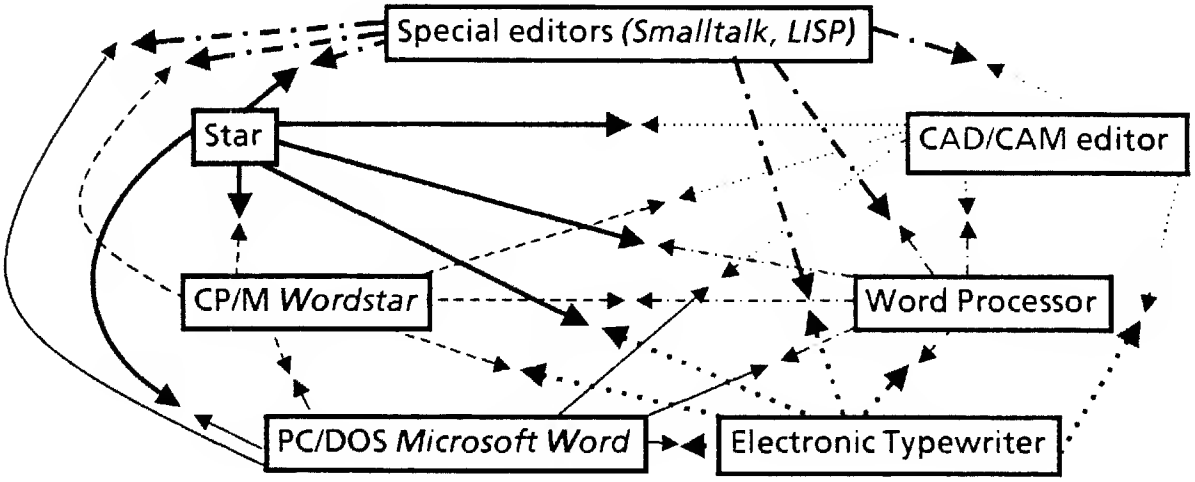

Figure 1.3: Required conversion routines for seven products.

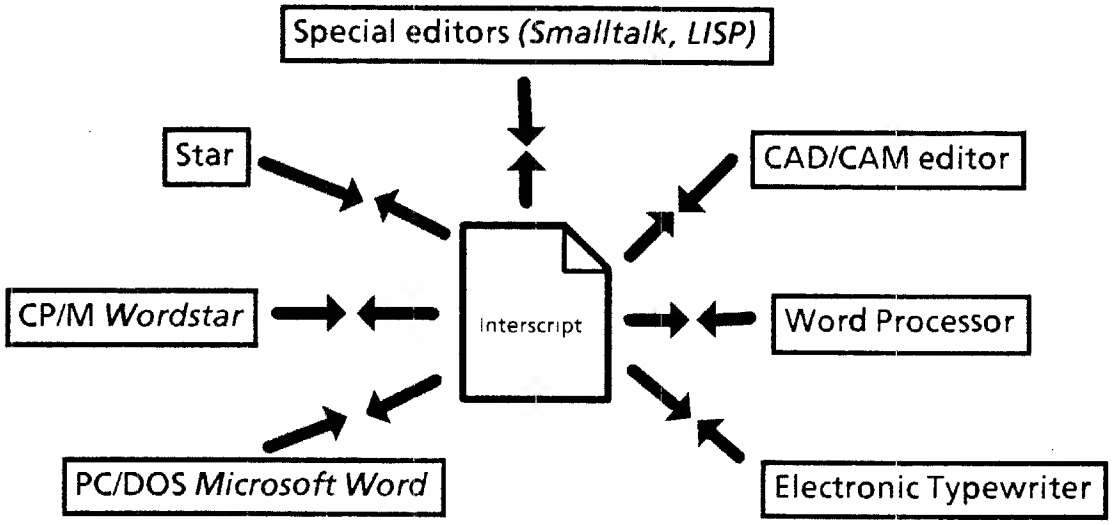

Figure 1.4: Required conversion routines using Interscript.

like the figures in this document. Customers want document exchange without information loss. Clearly we must have a better solution than writing converters for every possible permutation of document creation devices to be found in modern day offices.

1.2 The solution

One way to reduce the number of conversion routines is to provide a single, "all-inclusive" format. Only one routine is required to convert a given product's internal document format into the "all-inclusive" format and one more to convert from the "all-inclusive" format to the internal document format. Interscript is the name of the "all-inclusive" document format being developed by Xerox. The impact this has on the number of conversion routines is shown in Figure 1.4.

Having a document interchange format like Interscript means only 14 conversion routines are needed for seven products. A new product needs just two conversion routines to interchange documents with other products that accept the Interscript format. Only two conversion routines are needed to support a single "foreign" format such as the Navy's Document Interchange Format (DIF) or the International Organization for Standards Office Document Architecture (ODA).

1.3 Why not Interpress?

Interpress is the *final form* document representation Xerox has adopted for encoding documents for printing on electronic printers (see Figure 1.5). It is Xerox' intent that its document creation products will produce Interpress "masters" and that all its electronic printer products will accept and print these masters.

Although Interpress is a document interchange format, it operates in only one direction (from a document creator to a document printer). Interpress "masters" encode the information that is to appear as "marks on paper." As good as this is, it is not sufficient for interchanging documents in *revisable form*. Here's why:

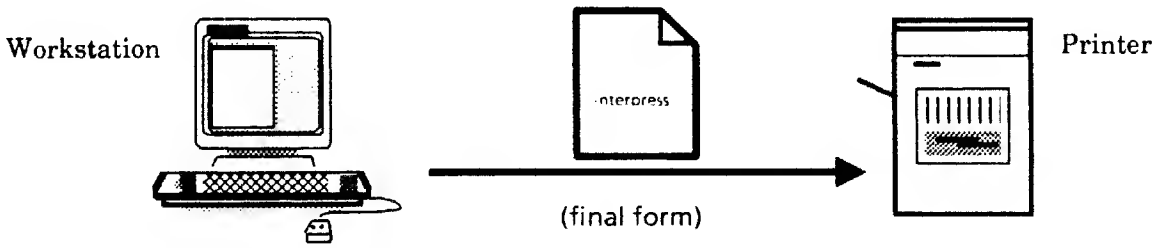

Figure 1.5: Interpress for documents in final form representation.

1. *Interpress has no conception of a document's logical structure.*
   A document might consist logically of front matter, body, and back matter. The body might consist of chapters, titles, sub-titles, sections, and paragraphs. These logical entities are important for subsequent editing and electronic processing. Interpress is only concerned about where to place "marks on paper."

2. *Everything has been specifically placed in an Interpress master.*
   Suppose one wanted to add a word to an Interpress master. That one word could change a line ending decision and even the page ending decision as one line moves from one page to another through an entire chapter. Interpress has no mechanism to alter the words in a master.

For these and other reasons, Interpress is not an appropriate format for interchanging documents that must be revised and edited.

1.4 Interscript's characteristics

Interscript is a kind of computer language for representing the content part, logical structure, and layout structure of documents (see Figure 1.6). All three are important.

Content part: What you see when you look at a document.

Logical structure: The way a document is organized, typically but not always in a hierarchical structure (paragraphs subordinate to sections that are subordinate to chapters).

Layout structure: The way a document is to be rendered (headings and footings on every page, multi-column dimensions, margin settings, and the like).

Interscript version 1.0 describes and accomodates the interchange of textual content, text layout, and page layout. Subsequent versions will describe other parts of documents that are not text (e.g. data-driven graphics such as pie charts, synthetic graphics built with points and lines, scanned images, spreadsheets [relationships among cells must be specially described], and the like).

The greatest challenge for any interchange format is to provide enough "richness" to handle every conceivable document. Interscript provides a base language that all Interscript accepting editors (and format converters) understand, called "Layer 1." The base language will change only when new basic objects (such as digitized audio, scanned images, or animated graphics) are defined. Editors that support an older version of the Interscript base language will still be able to co-exist with newer versions, though they will not be able to display or edit information described in new basic objects.

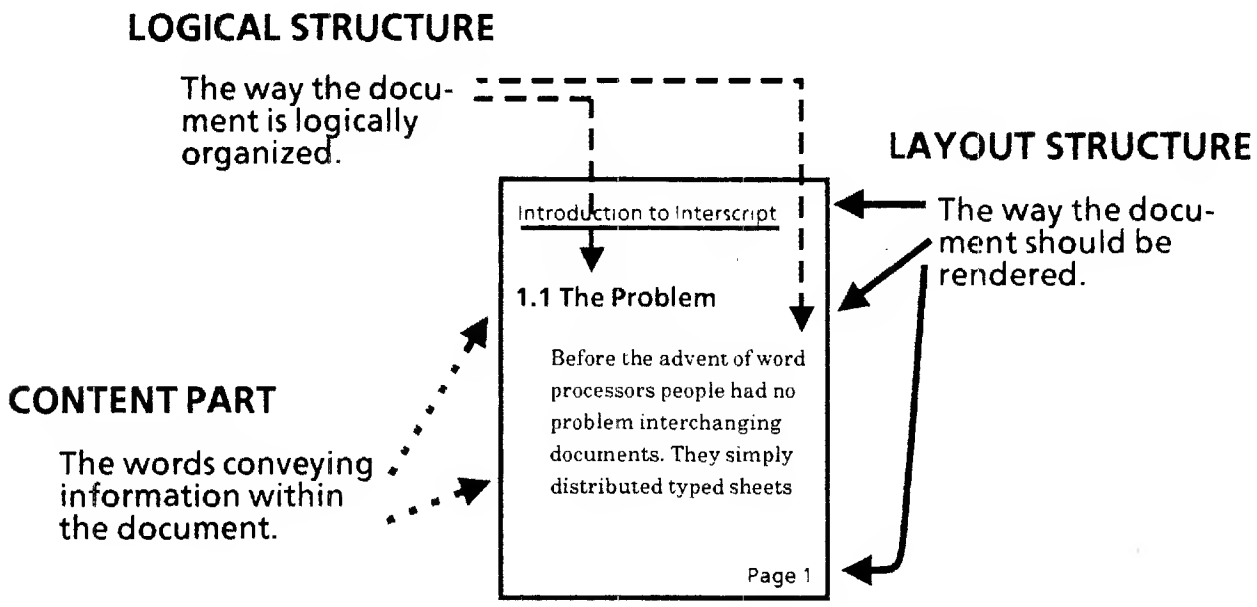

Figure 1.6: A document's constituent parts.

Layer 2 consists of certain common constructs such as "text," "box," "line," and "page." Most programs that process scripts (the term for a document encoded in Interscript) must implement most of Layer 2 if they expect to support the editing and rendering of textual information.

Some editors run on limited hardware which leads to problems with complex documents. An electronic typewriter does not really support multiple type faces (few operators are willing to swap type elements or print wheels when pages are being typed). Electronic typewriters certainly don't support multiple type *sizes*. What should a format converting routine do when it finds text that cannot be displayed on an electronic typewriter? There are at least two options: 1) Convert the document on a "best efforts" basis and discard whatever the electronic typewriter cannot represent; 2) Convert inappropriate constructs into text strings that can be displayed (markup codes). In all probability, users will be able to select the option that best suits their needs when documents are converted from one format to another.

A "best efforts" conversion is appropriate when a user knows a document will never be sent back to the originator. Consequently a word originally in italics could be emphasized by being printed in bold letters; tall letters can be printed in the normal font; unexpected characters, like Cyrillic, can be discarded. On the other hand, a conversion that does not lose information is necessary when a document will be returned to the originator or sent to another software editor. Figure 1.7 shows a document with markup codes (left-side) and how that document might be printed (right-side).

Although operators could be confused by markup codes, they will never "edit" them. Markup codes are not as understandable as a WYSIWYG (what-you-see-is-what-you-get) interface. (However it is easy to recognize text in the left box of Figure 1.7 that could be changed.) If text with markup codes is printed on an electronic typewriter, the pages will NOT be formatted correctly. But when a "marked-up" typewriter-like document is reconverted into a script (called externalizing in Interscript terminology) and then converted from that script to the format of a full-featured editor (called internalizing), the document

```
<PW,612/PD,792/CW,240/GW, 24/HYP>
<IWSP,72,25>
<RULE=HR,504,5.0,-24/VR,504,-24>
<#AA1=<4/CP>>
<#AA2=<4/QL,504,10>>
<#B2=<1/JU,240,0/WS,2,2,3/SP,2>>
<CW,504/NS/SP,18><AA2>SECTION 8:
<AA1> POLICY VALUES
<RULE><SP,24/CW,240><NS><Z>8.1
<B1> DEFINITION OF NET CASH VALUE
<b2>The net cash value is the
policy cash value plus the cash
value of any dividend additions
or accumulations, less any policy
loan interest. The policy cash
value is determined according to
the Basis of Computation Provisio
```

SECTION 8: POLICY VALUES

8.1 DEFINITION OF NET CASH VALUE

The net cash value is the policy cash value plus the cash value of any dividend additions or accumulations, less any policy loan interest. The policy cash value is determined according to the Basis of Computation Provision.

OPTION 1. LONGER TERM. You may continue your policy as extended term insurance by applying policy from the due date of the premium in defaut for a reduced amount of paid-up life insurance eligible for dividends. The amount of insurance will be that which the net cash value will purchase as a new single premium at the Insured's attained age on the due date of the premium in default.

OPTION 3. CASH

Figure 1.7: Document with markup codes and as printed.

can be formatted and printed correctly. There is always a benefit to using a low-level editor such as found on an electronic typewriter to make small editing changes.

Some diskette-based products like electronic typewriters have limited internal storage. The bits in a single scanned image could overwhelm the available diskette space if sent as a long string of [unintelligible] characters. In these cases a conversion routine might simply send a pointer to the unrepresentable construct (e.g. a scanned image) in a markup code. When the document in markup format is later externalized, the unrepresentable construct from the original script would be inserted when the markup code is encountered. (That code could, of course, be deleted by an operator, and that would delete whatever that code represented.)

Alternatively a conversion service might send a low-level product just the text fragments of a document with markup codes identifying where these fragments are located in the script. When these [edited] fragments are returned. a conversion service could recombine them with the information that describes the logical and the layout structure of the document.

These are a few ways a low-level editor could edit the parts of a document it understands while preserving those parts it does not understand. Note that this capability is available *without changing existing editors in any way*. This is an important consideration since it will take time before the world's present generation of text editors is superceded by a new generation.

2

Interscript overview

2.1 Some terminology

Documents are used to communicate information. This information is conveyed to people after being laid out on a plane surface (e.g. a terminal display screen or paper). Documents are laid out according to rules varying with cultures and languages. For example, western languages expect pages to be laid out from left to right and top to bottom. Such rules lead to a layout structure when a document is printed: text is placed in lines of a particular length, lines are assembled into paragraphs, paragraphs are placed on pages without isolating the first paragraph line at the bottom of a page, etc. The layout structure precisely defines where information is to appear on a surface when it is displayed.

Pieces of information within a document are usually organized to aid understanding. A document may be divided into chapters, chapters into sections, and sections into paragraphs. A chapter has a title and may have a subtitle. This structure is called the logical structure. There is a connection between the logical and the layout structure: the layout structure facilitates the understanding of the logical structure through typographic effects.

In order to interchange documents, computer systems must be able to specify the representation of documents as sequences of digital values. Document representations are expressed in terms of abstract constructs. Well-known constructs are pages, titles, paragraphs, etc. A document architecture specifies a set of abstract constructs, their meaning and their relationships.

If documents are to be interchanged among computer systems, those systems must have a common understanding of some particular architecture (i.e. the constructs that constitute the document representations). A document interchange standard provides such a document architecture. Computer systems may then build and circulate document representations using constructs defined by the standard.

The layout structure is constructed by the formatting process. The formatting process applies a style to set the appropriate rendering parameters. A particular text string in a document might be tagged as **emphasized** and the style may tell the formatter to use **bold** typeface for emphasized text strings. A formatting process needs to find layout information in a revisable form document to produce a formatted document. This infor-

mation should be represented by descriptions concerning pieces of information rather than being "hardwired" into a document as is typically the case with markup codes.

In some applications documents might be considered to have no layout characteristics. This would be true for an information retrieval system that searches filed documents for the presence of keywords. Such applications deal exclusively with text strings within documents and ignore layout. They may even ignore irrelevant parts of the logical structure, such as scanned images.

Some applications use a document's logical structure, yet also need to obtain layout information. When editing a document, an author needs to refer to both logical and layout structure in phrases like "see the footnote on page 17" or "as explained in section 4.3.2." The exact reference depends on how the final document is formatted.

The division between the logical and layout structure corresponds to the document production process, which has existed for only a few centuries. Today electronic documents are manipulated by many systems other than document production systems. Those systems have their own structured view of a document. Consider a computer system using forms. A form is a document and contains data such as default values, minimum or maximum values for numeric fields, and possibly some calculation formulas. These data are encoded as part of the form document, not in the software processing the form. Hence they are considered by this software as *contents* of the document, even though those data are not part of either the logical structure or the layout structure.

Electronic documents are different from paper documents. They may be manipulated (processed) by different applications for different purposes, yet each document is a single consistent unit. Any process operating on a revisable form document should always be able to find and store the data it understands into the structures it recognizes within the document. Hence one requirement for a revisable form standard is that it provides a mechanism to represent any kind of structure inside a document. More accurately, it should provide the capability to distinguish for each structure its *contents* and its *properties*.

The Interscript standard is a revisable form digital representation of documents to be interchanged among different computer programs or editors. The Interscript standard does allow the representation of structures together with their contents, their properties, and associated descriptions. But there are many other goals that have influenced the design of Interscript.

2.2 Layer 1: The Interscript base language

Conventional systems for the representation of data structures in a data stream provide a set of syntactic rules about the placement of data and operators in the stream. Whether the syntax is a static file format or more complex, a language is needed. Hence the Interscript base language defines the representation of documents in a byte stream. This digital representation encoded using the Interscript base language is called a script. Note that it is an encoding of the document, not the document itself.

Even though there is no international standard for document interchange, computer users have had some success in limited interchange. A few interchange formats have been markup languages such as TEX [Reference 1], Scribe [Reference 4], and TROFF [Reference 5]. These markup languages were originally provided to achieve several goals:

- Allow users who did not understand typography to obtain good quality printing;
- Make it possible for users without high-quality terminal displays to produce documents requiring complex layouts and different typefaces;
- Allow a document to be printed on different printers.

Though markup languages were not designed as interchange formats, marked up documents are revisable to some extent. One can enter new markup signs to change the layout or modify a document's content and continue to get the same layout. The same document can also be printed on different computer systems with different printers.

The flexibility provided by markup languages comes from the fact that documents are parsed and interpreted. Programs parse the documents and associate some meaning to the markup symbols they encounter. They then interpret the information found in the document according to the symbol's meaning. The markup sign TITLE, for example, might be used to interpret the layout of the following character string according to both a style and the specified printer.

New markup signs may be added to the language, which allows for an infinite extension of possible interpretations. However one must describe the meaning of new markup symbols to have a precise and "communicable" understanding of those symbols. A language provides the formalism. The appropriate language to describe the meaning of the markup symbols is clearly NOT the markup language itself. Rather it is the meta language of the markup language. Many markup languages do not have a formal meta language. They usually hard-wire the meaning of markup symbols into the parsing software. A kind of macro facility is sometimes provided to rename a set of markup signs as a single sign to make it easier to use. This makes it very difficult to translate a document from one markup language to another, and makes it impossible to input a TROFF document, say, into a Scribe parser.

Unlike an object manipulated by a software application—which is concerned only with the abstractions developed for that application—a single document may be manipulated by many different applications. Although many processes operate on revisable documents and usually change the data, they must not affect any data not concerned with the operation. For example, some editing systems can manipulate bar charts or equations; others cannot. Editing systems with different capabilities must retain the constructs they find in a document even though they may not be able to display or edit those constructs. These are fundamental principles in the Interscript design:

- Any operation is expected to process known constructs and preserve unknown constructs without change. An Interscript editor must not lose information when processing a script.
- Any operation must control the validity of its own actions. For instance, some construct may be restricted to contain only "paragraphs." Then an application must be sure to put only paragraphs into this construct.

Layout and logical structures are basically tree structures. This property makes it easy to have them mapped onto a data stream through syntactic markers. However sometimes it is necessary to maintain other structures. When a phrase as "see paragraph xx" is encountered, xx may be unknown. One must then maintain a link between the source

paragraph and its reference such that they remain consistent. It may be that the linked elements do not belong to the same structures as "see figure xx on page yy." These links cannot be explicitly introduced into the data stream tree structure. A binding is required that dynamically links the necessary objects.

Interscript provides a comprehensive and simple binding mechanism. A binding associates a name with some value; the value is then accessed through the name. This very general operation may also be used to achieve compactness. A form might contain, for example, the same item repeated 25 times. A compact script would bind it to a name and repeat that name 25 times.

The part of the Interscript standard concerning the base language is called Layer 1 of the standard. In addition to the base language, the Interscript standard also provides a set of abstractions that are expected to be widely used. These abstractions are described using the Interscript base language. Since those constructs have their own significance and are built on the top of Layer 1, this part of the Standard is called Layer 2. Convenient abstractions can be designed for a particular application in terms of the Interscript base language. New constructs can and will be added to Layer 2 as necessary by those responsible for maintaining the Standard.

2.3 Layer 2: The layout model and standard constructs

Structuring a document into its logical components identifies common structures that should be handled in a common way. These logical components require layout rules specifying how each should should be rendered on a page. Interscript's layout process considers logical components of a document as liquid that is poured onto a surface. This pouring operation involves two kinds of entities: the liquid that is poured (typographers have called this the galleys) into rectangular shapes (called boxes) constructed according to rules given by a template.

Most of the standard constructs in Layer 2 concern layout. This is because the interchange of electronic documents has not yet reached a point where standard non-layout constructs exist. Few organizations use the same typical logical structures. Although people use terms like letter, memo, or report, these terms actually describe many different document types. It is therefore premature to introduce standard constructs for those terms.

2.4 Other Interscript traits

A representation standard must be extensible to allow for future requirements. Interscript provides extensibility through the base language. The Interscript version 1.0 base language contains basic objects that are needed to describe ordinary documents (e.g. atoms [names] and integers [characters are considered integers]). When the set of basic objects is enlarged, the version of Interscript must be increased. The types of information contained in documents will change slowly. Today's documents contain numbers, characters, and sometimes scanned images; tomorrow's electronic documents will contain digitized sound and perhaps digitized video.

Since scripts are document representations to be interchanged among machines, a script in its machine encoding is about as readable as a program in binary code. Because only computer systems will directly manipulate scripts, the Interscript base language was

not designed to be manipulated by people. A format has been designed so human beings can read documents containing Interscript examples, but it is still difficult.

The Interscript Standard defines how to represent a document as a sequence of bytes. It makes no attempt to standardize how the byte sequence of a script should be transmitted or stored.

The Interscript Standard provides a vehicle for document interchange. It does not provide an efficient internal encoding to be used in the memory of a computer system.

The Interscript Standard does NOT describe a particular set of functions that could be available in a system. Nor does it describe or give clues about the user interface of an Interscript-based system.

The Interscript base language is independent of standard constructs that are added, removed, or modified within the Standard. If the syntax of the base language is changed during the standard's life, changes in the description of constructs will occur. But the concepts behind those constructs will remain. Once a paragraph, always a paragraph.

3

Base language: syntax

This chapter describes the Interscript Base Language syntax (the symbols that may appear in a script and the order in which they must appear).

3.1 Publication encoding and machine encoding

The Interscript base language defines the encoding of scripts. A script is exchanged among computers and consists of a sequence of digital values. The syntax used by computer systems to interpret these values is called the machine encoding. It is designed for convenient processing by computers. The syntactic definition of the machine encoding will be published in early 1986.

The best way to represent data for computers is usually not a good way for people. If, for example, an Interscript example were presented as a sequence of encoded bytes, it would be very accurate. It would also be very tedious for people to understand. Consequently we use another syntax to communicate scripts among human beings. This syntax encodes a script as a series of text characters and is called the publication encoding. The numeric value one hundred and twenty three, for example, is represented in the publication encoding by the three-character string 123; in the machine encoding, a binary value is used. It will be easy to convert any script from one encoding into the other. The publication encoding will be used as an interchange format among Xerox systems during 1985.

All examples in this document use the publication encoding whose syntax is described in this chapter.

3.2 The Interscript base language

A document is a collection of structures. Consequently there is a need to refer to structures or parts of structures within documents. For example, one might like to give a name to the description of a structure that usually contains a number of sentences and refer to that description by a name, like "paragraph," say. Interscript provides a mechanism to bind values to names, much like a variable takes on or "is bound to" a value in a computer program.

Standard programming languages could be used to express those document structures. To get the flavor of documents written in a programming language, Example 3.1a shows a simple document that is written in Pascal in Example 3.1b.

Example 3.1a: A simple document.

Title of document

Paragraph 1 is made out of this single sentence.
This second paragraph is the last paragraph of the document.

Example 3.1b: The simple document expressed as a Pascal program.

```
Program Document;
   {In this example we assume the Text type exists. It has two fields: chars
   containing characters and font indicating a font. We also assume a
   paragraph consists of only one text unit}

TYPE

   ParagraphList = ↑ Paragraph;
   Paragraph = RECORD
      flush : (left, centered, right);
      text : Text;
      next : ParagraphList;
   END;

   Document = RECORD
      contents  : ParagraphList;
      reference : Integer;
      author    : String;
   END;

VAR

   me : Document;
   parList: ParagraphList;

BEGIN
 me.reference := 125782;
 me.author := 'Vania';
 new(parList);                          { make title }
 with parList↑ do                       { set title }
 begin
    text.chars := 'Title of document';
    text.font.typeface:= bold;
    flush := centered;
    new(next)                           { make paragraph one}
 end;
 with parList↑.next↑ do                 { set paragraph one}
 begin
    text.chars := 'Paragraph 1 is made out of this single sentence.';
    flush := left;
    new(next)                           { make paragraph two}
 end;
```

```
 with parList↑.next↑.next↑ do        { set paragraph two}
 begin
    text.chars:='This second paragraph is the last paragraph of the
document.';
    flush := left;
    next := NIL
 end;
 me.contents:= parList ;
 repeat                                  { assign font TimesRoman everywhere }
    parList↑.text.font.name:= TimesRoman;
    parList := parList↑.next
 until parList = NIL
END.
```

The Pascal program in Example 3.1b is a valid representation of the example document in Example 3.1a. When the program is executed, it creates a data structure in the memory of the computer reflecting the structure of the document (a title and two paragraphs).

If we used Pascal as a document representation language, we could define document constructs like paragraphs. This would guarantee a common, precise understanding among computers of what constitutes a paragraph. There are, however, major disadvantages with using Pascal for such a representation (e.g. Pascal does not support variables of unknown type, it creates a large overhead, etc.). These disadvantages make using Pascal as a document representation language inadvisable.

Interscript defines its own mechanism to describe data structures that is particularly suited for the description of constructs occuring within documents: the Interscript base language. Since that's like many programming languages, we could say the Interscript base language is a kind of *programming language*.

The units that are interchanged between Interscript systems are called scripts. When being transmitted, documents must be expressed as scripts. Considering the Interscript base language to be a programming language suggests what may be the most important concept in Interscript: a script is not a straightforward mapping of a document structure into a byte stream: it is a *program in the Interscript base language*.

As with Pascal programs a recipient must execute an arriving script and transform it into an internal structure. This internal structure will mirror the original script, with some evaluations performed. Local applications, especially the editor, will read and manipulate it. Producing this internal structure is the task of the internalizing process. When transmitting documents, a sender must transform them into scripts. This is the task of the externalization process. Figure 3.1 illustrates Interscript's processing model.

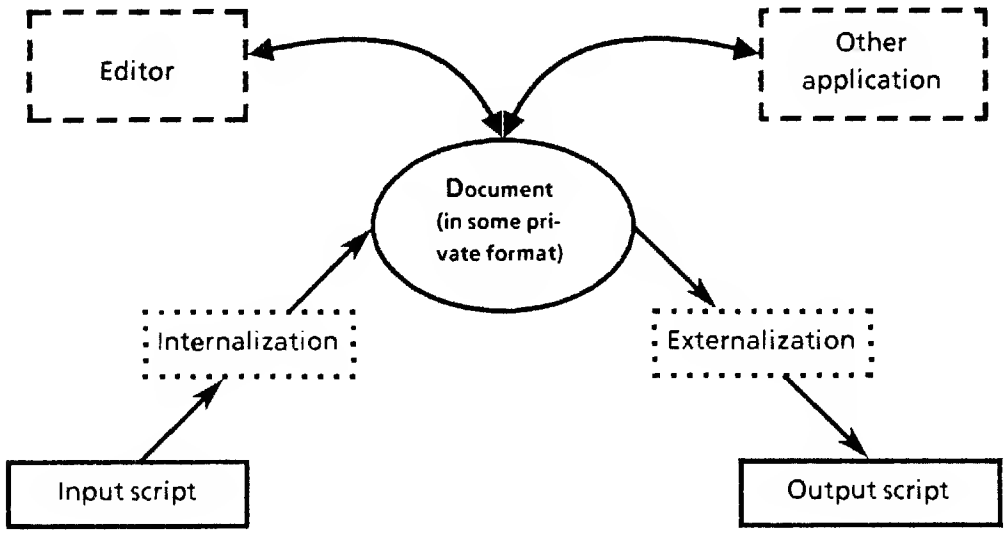

Figure 3.1 Interscript processing model.

3.3 Interscript objects

The entities occuring in a script are called Interscript objects. This section provides an overview of the fundamental objects and their properties.

3.3.1 Scripts

Many programming languges use the bracketing of expressions to encapsulate smaller units and to indicate an appropriate place in a superior unit (e.g. Pascal uses *begin* and *end* keywords, Lisp has parenthesized expressions). Interscript has a similar bracketing notation using braces ({ }). The units these braces enclose are called nodes. Like blocks in a programming language, nodes delimit the scope of the constructs they contain and build-up tree-shaped structures.

A script presents itself as a tree of nested nodes forming a hierarchy called the dominant hierarchy. Such a dominant hierarchy may either express the logical or layout structure of a document. Not all nodes appearing in a script belong to the dominant hierachy. Some specify values for bindings (see below); such nodes are associated with names and are not subnodes of the superior node.

3.3.2 Nodes and tags

Consider a bottle of wine. It consists of two parts: a container and contents. The container is independent of contents. In fact, for special display purposes there may be no contents at all.

Tags are affixed to the container. Some are mandatory, like the percent of alcohol by volume and the capacity of the container. Some are highly desirable, like the product number encoded in wide and narrow bars affixed to the bottle by the manufacturer and

the price affixed by the retail store. Some are optional, like words of praise from a long forgotten state fair. With imported wine, some consumers may not understand everything on a tag because it is written in a foreign language. But even so, one can open the container and enjoy the contents.

The nodes in a script may be compared to bottles of wine. Like bottles of wine, nodes are containers with contents and tags.

Since scripts are programs, the content of a node is known only *after* the internalizing process has evaluated all "expressions" and "formulas" within the node. The resulting content objects, if any, may be basic objects like integers or atoms, subnodes, or structural bindings. They all are "structural," i.e. the place they occupy within a node is important and must carefully be preserved by Interscript systems. Removing a content object changes the whole internalized script (basic objects represent the text portions of a document). The content objects are sequentially ordered as they appear—from left to right—in the superior node. Note that tags and non-structural bindings do NOT count as content objects.

A node may or may not be accompanied by one or more tags. Tags are Interscript objects providing information about the node. A list of attributes is associated with most tags to specify information. Tags are used for three purposes:

- Tags may make some categorical statement about the content (e.g. the CHAR$ tag requires the content to be an integer and it is interpreted as a character). Such tags are called *content tags*.
- Tags may provide additional information related to a node's content (e.g. the LABEL$ tag supplies a set of labels).
- Tags may create record-like structures that are self-contained and not related to the node content. Such a "record" consists of values for the tag attributes (e.g. nodes with the MEASURE$ tag provide a triple of numbers attached to the three MEASURE attributes). Such tags are called *record tags*.

Instead of saying "that node carries the tag T$," this document uses the expression "that node is a T node."

When evaluating a node, the internalizing process builds a list of tags and attaches this list to the internalized node (at least in this version of Interscript). Therefore tags do not "remember" their place within a node.

3.3.3 Basic objects

Interscript version 1.0 has these basic objects: Atom, Boolean, Integer, and Number (also called basic types). Each basic object is considered a built-in Interscript object that is not formally defined in the base language. It provides an encoding-decoding mechanism for transforming basic objects into binary values and for interpreting binary values as basic objects (e.g. a particular byte sequence may be interpreted as an integer).

Basic objects constitute the contents of "leaf" nodes (those are the nodes containing no subnode). In publication encoding they are characterized by special rendering properties:

- Basic objects of type Atom appear as strings encoded in the ISO646 character set. They contain only letters and digits. They are used as (machine readable) names and identifiers.
- Basic objects of type Boolean appear as TRUE or FALSE (these are reserved keywords).

- Basic objects of type Integer and Number appear as decimal values expressed in digits. As usual, Integers and Numbers may be positive or negative; Numbers may have fractional values.

Note that there is NO basic object of type String. In Interscript, strings are expressed as nodes with a special tag (the CHAR$ tag) containing an integer. The integer encodes a particular character. To make reading easier, the publication encoding presents these integers as characters enclosed in angle brackets (see section 7.3).

3.3.4 Bindings and references

As in programming languages, it is often necessary in Interscript to attach a name to a value. Such named values may be copied to different places by referencing the name. Of special importance are names denoting tag attributes that are bound to the corresponding attribute values. The bindings in Interscript fulfill this task.

Unlike most programming languages, Interscript has a rigorous naming concept. A name does not automatically refer to the value to which it is bound. An explicit construction, called a reference (denoted by #), must be used to get the associated value. When the internalizing process encounters a reference, it starts a look-up procedure for a binding with a matching name. This look-up procedure goes upward in the dominant script hierarchy ("from right to left" if the script is considered to be written as a linear sequence). Note that all nodes encountered this way are already opened and not completely internalized. The tags and bindings these nodes provide are called the actual environment.

Interscript supplies two kinds of bindings: non-structural and structural. When the internalizing process reaches the end of a node, it eliminates non-structural bindings (denoted by =) from the content. If the non-structural binding provides a value for a tag attribute (such bindings are called relevant), it is put into an attribute list; otherwise it is discarded. A subsequent non-structural binding with the same name overwrites the last one encountered.

Structural bindings (denoted by =%) are part of the content and preserve their position within the node. In particular, several structural bindings with the same name may appear in an internalized node.

3.3.5 Formulas

Consider this problem: a document contains a title and text paragraphs with the rule that the (left) margin of the text is six units indented compared with the title margin. How is this rule to be expressed in Interscript? The simple solution utilizing non-structural bindings and references does not work; e.g.

```
TitleParaMargin = 10  TextParaMargin = TitleParaMargin# + 6
```

The internalizing process will calculate the correct value 16 for the text margin; but after this calculation, TextParaMargin is bound to 16 and its dependence on TitleParaMargin is lost.

To prevent this loss, Interscript provides formulas (denoted by %). Formulas require that the original expression be kept with the result to allow recalculating the value of the formula (for example, when a user changes the title margin to 12, the text margin should change to 18).

Formulas are the most difficult part of the base language since they involve several problems and subtleties. In the TitleParaMargin example, it is not sufficient just to bind TextParaMargin to a formula:

```
TitleParaMargin = 10  TextParaMargin = (TitleParaMargin# + 6)%
```

The internalizing process will put both bindings into the attribute list (assuming that both bindings are relevant) destroying the order in which these bindings appeared. After internalizing, the reference to TitleParaMargin is inappropriate; it is not clear which binding of TitleParaMargin should be used (particularly if a second binding of TitleParaMargin overwrites the first). Therefore formulas must preserve:

- the expression that provides the result of the formula;
- the positions of all references that occur within the formula;
- the position of the formula itself.

The first requirement is met by using a formula, the remainder by using structural bindings. To emphasize the common requirements to remember positions, the **%** symbol denotes formulas as well as structural bindings. The correct solution to our problem is therefore:

```
TitleParaMargin = % 10  TextParaMargin = % (TitleParaMargin# + 6)%
```

3.4 Base language syntax for the publication encoding

This section describes the publication encoding syntax for the Interscript base language. The grammatical rules are explained from a *syntactical* point of view, i.e. what kind of entities may appear in a script and in what order. The rules enable the internalizing process to identify these entities and activate corresponding operations when parsing a script. The *semantics* of these entities, i.e. the definition of the corresponding operations, is discussed in Chapter 4.

The syntactic rules are designed to support fast parsing. The syntactic rules allow constructing syntactically legal scripts that will be rejected by the semantic operations. The syntax does not contain rules about how a script is displayed in publication encoding. Most of the examples in this document will contain spaces, rendering characters such as line feed or tab, and different fonts. The usage is considered self-explanatory.

The formalism used to define the base language is Backus-Naur Form (BNF). The BNF description of any formal language comprises a series of replacement rules called productions. By following the replacement rules, all valid expressions of the language can be generated. Three classes of symbols appear in a production: terminals, non-terminals, and operators. A terminal is a symbol that appears literally in the language being described. Terminal symbols are defined in the productions as a string within double quotes ("). A non-terminal is a symbol that is defined as equivalent to either a particular series of symbols—either teminal or non terminal—or a choice among several such series or denotes a basic object that is not defined in the syntax. Such basic objects are expected to be well-known in the environment interpreting the syntax.

Two operators are used in productions. The equivalence operator (:: =) defines the replacement rule for the non-terminal appearing on the left side. The alternative operator (|) distinguishes among several possible alternatives, one (and only one) of which must be selected. When several options are offered within an alternative, they are separated by an alternative operator and grouped within parentheses.

The ***empty*** symbol appearing in the right side of a rule is a special descriptor meaning the left side non-terminal may be replaced by nothing.

The entire grammar for the Interscript base language can be summarized in only a few rules. Here they are:

```
Rule 1   script :: = "INTERSCRIPT/1.0"  node  "ENDSCRIPT"
Rule 2   node :: = "{" items "}"
Rule 3   items :: = empty | items item
Rule 4   item :: = tag | expression | nonStructuralBinding | structuralBinding | formula
Rule 5   tag :: = atom"$"
Rule 6   expression :: = term | operation term | expression operation term
Rule 7   operation :: = " + " | "-" | "*" | "/" | "!" | "|" | "<" | ">" | "< =" |
                                                  "> =" | "equal" | "not" | "or" | "and"
Rule 8   term :: = basicObject | reference | node | "(" expression ")"
Rule 9   basicObject :: = Atom | Boolean | Integer | Number
Rule 10  reference :: = (name | path | reference)"#"
Rule 11  name :: = Atom | qualifiedName
Rule 12  qualifiedName :: = Atom "." Atom
Rule 13  path :: = reference ":" qualifiedName
Rule 14  nonStructuralBinding :: = name " = " (expression | quotedExpression)
Rule 15  structuralBinding :: = name " = %" (expression | formula | quotedExpression)
Rule 16  formula :: = expression"%"
Rule 17  quotedExpression :: = """ expressionList """
Rule 18  expressionList :: = expression | expressionList expression
```

Rule 1

```
script :: = "INTERSCRIPT/1.0"  node  "ENDSCRIPT"
```

In the publication encoding, a **script** is defined as the character string within the double quotation marks, one node, and a second character string within the double quotation marks, in that order. (Of course a script constructed under a subsequent version of Interscript would be identified by a version number greater than 1.0.) The presence of this specific character string insures that this is really a script. The version identifies the encoding being used. The specific trailing character string at the end of the script insures the correct balancing of braces in the dominant hierarchy.

A **node** is defined in the next rule. The one appearing in rule 1 plays a particular role. It is the outermost node in a script and is called the root node of the dominant hierarchy.

Rule 2, 3

```
node :: = "{" items "}"
items :: = empty | items item
```

Rule 2 says a node is defined as a left brace, something called **items**, and a right brace.

Rule 3 says **items** is a list of zero or more entities called **item**. Hence a node is a list of items between braces. Example 3.2 is the simplest possible valid script that can be designed, the empty script.

Example 3.2: The empty script.

```
-- 1 --    INTERSCRIPT/1.0 { } ENDSCRIPT
```

Rule 4

```
item :: = tag | expression | nonStructuralBinding | structuralBinding | formula
```

An **item** is defined as one and only one of the enumerated alternatives, all of which are defined in subsequent rules. Items provide the tags, contents, and bindings that constitute a node.

Rule 5

```
tag :: = atom"$"
```

A tag is defined as an atom followed by a dollar sign. A tag describes the node containing it. Example 3.3 contains a one node script (because it has only one set of braces). This node contains a tag, CHAR$, that identifies the content of this node as one or more characters.

Example 3.3: A simple script containing a character string. †

```
-- 1 --    INTERSCRIPT/1.0
-- 2 --    { CHAR$ <This is a one node script. It contains a string.>}
-- 3 --    ENDSCRIPT
```

Rule 6

```
expression :: = term | operation term | expression operation term
```

Expressions provide that part of a node's content whose derivation is NOT "remembered." They are evaluated by the internalizing process and replaced by the result. They may be a single operand, called a **term**, or combine values according to the **operation** specified, producing a new value.

Rule 7

```
operation :: = "+" | "-" | "*" | "/" | "!" | "|" |
               "<" | ">" | "<=" | ">=" | "equal" | "not" | "or" | "and"
```

The arithmetical operations allowed within expressions are addition (+), subtraction (–), multiplication (*), and division (/). Although with this formalism any term may be

† The examples in this chapter are greatly oversimplified. Much has been omitted to avoid confusion.

involved, the semantic actions associated with these operations accept only Integer and Number values as operands.

The index operator (!) provides access to a particular content object, i.e. a subnode, basic object, or structural binding (the node's content is sequentially ordered). Normally the index operator is used to access unnamed content objects (subnodes and basic objects). Semantically the expression on the left-hand side of the ! must be a node and the right-hand side be an integer used as an index. In Interscript, the index value of the first item in a sequence is zero.

The opening operator (|) is used to get *all* the content of a node. The semantic action requires the expression preceding the | to be a node.

The relational operations are the strict and inclusive numeric comparison (**<**, **<=**, **>**, **>=**) and identity (**equal**). They produce Boolean values. Except for identity, the semantic actions allow only Integer and Number values as operands.

The logic operations are negation (**not**), conjunction (**and**), and disjunction (**or**). The semantic actions allow only Boolean values as operands.

Between the above operations precedence rules are enforced (e.g. multiply and divide occur before add and subtract). Parentheses may be used to force one calculation to occur before another.

Example 3.4 Syntactically legal operations.

```
--1--   1 + {1}                      result: error
--2--   4 / 0                        result: error
--3--   {1 2 3 4} ! (1 + 1)          result: 3
--4--   {CHAR$ <text1> <text2>} |    result: <text1> <text2>
```

Rule 8

```
term :: = basicObject | reference | node | "(" expression ")"
```

Terms are the operands in expressions. They may be basic objects, references, subnodes, or expressions contained within parentheses.

Rule 9

```
basicObject   :: = Atom | Boolean | Integer | Number
```

As discussed in section 3.3.3, Interscript has Atom, Boolean, Integer, and Number as basic objects. They are not defined in the base language.

Rule 9 is the only syntax rule that will change with new Interscript versions as new basic objects are incorporated into the base language.

Example 3.5 Leaf nodes containing basic objects.

```
--1--   {A  B}          Atoms
--2--   {A = B  TRUE}   Booleans
--3--   {1 2 3}         Integers
--4--   {11.2 + 2}      Number
```

Rule 10

```
reference ::= (name | path | reference) "#"
```

References point to bindings whose values will be copied by the internalizing process replacing the reference. The referred binding is the "first" one that is encountered when the actual environment is scanned upward for a matching name. Such a binding may be structural or non-structural.

Syntactically, references are denoted by the # symbol that follows directly after a name or path identifying the desired binding.

References may be used recursively to allow indirect addressing. Indirect addressing is like calling information to get a telephone number. The "address" of information is known, and therefore the "address" of someone else can be obtained. Indirect addressing in a computer means using what is contained in a known location as a pointer to a data element. Scripts cannot explicitly use pointers, *per se*, as pointers depend upon memory mapping. A script may, however, bind a name to another name that produces a value when the reference process is applied twice. A multiple reference indicates the need to follow an indirect path to obtain a value. Example 3.7 shows such a double reference (in connection with bindings).

Rule 11, 12, 13

```
          name ::= Atom | qualifiedName
qualifiedName ::= Atom "." Atom
          path ::= reference ":" qualifiedName
```

These rules control naming in Interscript. There are:

- simple **names** consisting of an atom.
- **qualified names** consisting of two atoms separated by a dot. Normally qualified names are used to name tag attributes: the first atom is the tag's name and the second atom is the attribute's name.

  Simple and qualified names refer to bindings accessible within the actual environment (i.e. the hierarchy of opened nodes that are not yet internalized).
- **paths** consisting of a reference and a qualified name separated by a colon. The qualified name provides a tag attribute and the reference points to a node that should provide a relevant binding for the that tag attribute.

  Paths always refer to relevant bindings of internalized nodes that are not contained in the actual environment.

  Note that paths may only occur on the right-hand side of bindings. The values they point to belong to nodes that have been internalized and therefore are "read-only."

Example 3.6: Some references.

```
--1--   T.A#          Reference with qualified name
--2--   B##           Double reference
--3--   C#:T.A#:T.B   Nested path reference
--4--   (C#):(T.A#)   Not legal
```

Rule 14

nonStructuralBinding ::= name "=" (expression | quotedExpression)

A binding equates or "binds" a name to a value provided by an expression or to a quoted expression. In contrast to structural bindings, **non-structural bindings** are eliminated from the contents after the internalization process has evaluated them. They are used for two purposes:

- Non-structural bindings may save bits in the machine encoding by binding values to names and using references to those names. The internalizing process replaces those references by their common value and eliminates the binding. This usage is not of interest for the publication encoding.
- Non-structural bindings may provide values for tag attributes (relevant bindings). In this case the name must be a qualified name.

Example 3.7 shows how double references work using non-structural bindings. The goal of this node is to express that the margin text should be placed 10 units below the margin position. This margin is either the top margin or the bottom margin according to the value of the binding for the name where.

Example 3.7a: A double reference in a node.

```
--1--   {MARGIN$ CHAR$
--2--       MARGIN.TopMargin = 0
--3--       MARGIN.BottomMargin = 270
--4--       where = MARGIN.TopMargin
--5--       MARGIN.TextPosition = where## + 10
--6--       <text placed at the top or at the bottom of the page>}
```

Note that in this example where is bound to the atom MARGIN.TopMargin and NOT to the value of MARGIN.TopMargin which is 0. The first reference of where in line 5 supplies the atom MARGIN.TopMargin, the second reference supplies the value 0. Example 3.7b shows the corresponding internalized node.

Example 3.7b: The node of Example 3.7a internalized.

```
--1--   {<text placed at the top or at the bottom of the page>} with
--2--       the tag list ( MARGIN$ CHAR$ ) and
--3--       the attribute list ( MARGIN.TopMargin = 0
--4--                            MARGIN.BottomMargin = 270
--5--                            MARGIN.TextPosition = 10 )
```

An important process is referencing and opening nodes. A node may be bound to a name and later referenced and opened. An opening places the contents of the node into the environment in which it is referenced. Referencing and opening a node is a feature similar to the include of some programming languages, used to include some definitions, usually declarations, into a program. This Interscript feature may be used to introduce default values for tags or bindings anywhere in a script. Such a node reference can also be used to associate styles with parts of a document. A node may then be "imported" and locally modified because the node reference may be followed with additive bindings in the node where it is referenced. This is illustrated in Example 3.8a and b.

Example 3.8a: Referencing and opening nodes.

```
--1--   { CHAR$ FONT$
--2--       myFont = {FONT$
--3--                   FONT.Name = Helvetica
--4--                   FONT.Points = 10
--5--                   FONT.Boldness = regular }
--6--       myFont#|  FONT.Boldness = darker
--7--       < Here is the title > }
```

A node is (non-structurally) bound to the name myFont in line 2. It provides a default font for use throughout the node beginning on line 1 via a reference to myFont. The reference preceding a title on line 6 specifies that the title font comes from myFont (the document font), but is printed in bold face. The expectation is that if the font of the document text is changed (by changing lines 3 through 5), the font of the title will also change in the same way, yet remain bold. (See Example 3.9a for the correct way to meet that expectation.) The internalized node is shown in Example 3.8b.

Example 3.8b: The internalized node of Example 3.8a.

```
--1--   {< Here is the title > } with
--2--       the tag list ( CHAR$ FONT$ ) and
--3--       the attribute list ( FONT.Name = Helvetica
--4--                   FONT.Points = 10
--5--                   FONT.Boldness = darker )
```

Rule 15

structuralBinding:: = name" = %"(expression | formula | quotedExpression)

Like non-structural bindings, **structural bindings** attach names to values for later reference. In contrast to non-structural bindings, they are part of the contents and must be retained wherever they occur within a node. Since they remain in an internalized node, they can be referenced in formulas: the way a referenced value was obtained is reconstructible after internalization.

Consider the internalized node of Example 3.8b. The font style called myFont in the original node has disappeared although myFont might provide significant information to an editor. In Example 3.9a myFont is structurally bound (denoted by the = % in line 2) and therefore transferred into the internalized node.

Example 3.9a: Node with a structural binding.

```
--1--   { CHAR$ FONT$
--2--       myFont = % {FONT$
--3--                   FONT.Name = Helvetica
--4--                   FONT.Points = 10
--5--                   FONT.Boldness = regular }
--6--       myFont#|  FONT.Boldness = darker
--7--       < Here is the title > }
```

Example 3.9b: The internalized node of Example 3.9a.

```
--1--   { myFont = % {FONT$
--2--                 FONT.Name = Helvetica
--3--                 FONT.Points = 10
--4--                 FONT.Boldness = regular }
--5--   < Here is the title> } with
--6--       the tag list ( CHAR$ FONT$ ) and
--7--       the attribute list ( FONT.Name = Helvetica
--8--                            FONT.Points = 10
--9--                            FONT.Boldness = darker )
```

If a formula is bound to a name, a structural binding must be used. In particular, if the value of a tag attribute should be given as a formula, the corresponding relevant binding must be structural.

Rule 16

formula ::= expression"%"

When an expression is evaluated, it is replaced by the result. To retain the original expression with the result, a **formula** must be used. The original expression serves as a rule that can be applied to recalculate the result, perhaps after an editing operation.

Syntactically, a formula is characterized by the % symbol following an expression that must be remembered.

Semantically, a formula requires that the references it contains refer to structural bindings. Otherwise it would not be possible to recalculate the result of the formula. A special case are references using a path since two references occur. In this case only the first reference, which refers to the node, must be structural.

Formulas appear as values of bindings or as node items. In the first case they must be structurally bound and therefore are part of a content object (see rule 15). In the second case they are a kind of "ghost" content. On one hand, formulas do not count as content objects; their result belongs to the content. On the other hand, the position of a formula within a content sequence must be remembered for later recalculation.

Example 3.10 shows a typical use of a formula (in line 6 to compute the value of the text margin). It specifies that the margin for text nodes is to be the margin for titles plus 6 units (e.g. text margins are indented six units compared to a maximum-width title).

Example 3.10a: Defining a paragraph margin relative to a title margin.

```
--1--   {title = % {PARA$
--2--                 PARA.LeftMargin = 10
--3--                 PARA.Raggedness = centered
--4--                 PARA.Justification = FALSE}
--5--   text  = % {PARA$
--6--                 PARA.LeftMargin = % (title#:PARA.LeftMargin# + 6)%
--7--                 PARA.Raggedness = left
--8--                 PARA.Justification = TRUE}
--9--       { PARA$ title#|%
--10--                < Here is the title> } }
--11--      { PARA$ text#|%
--12--                < The first paragraph starts here> } }
```

Example 3.10b: The internalized node of Example 3.10a.

```
--1--   {title = % { } with
--2--                   tag list (PARA$)  and
--3--                   attribute list (PARA.LeftMargin = 10
--4--                                   PARA.Raggedness = centered
--5--                                   PARA.Justification = FALSE)
--6--   text = % { PARA.LeftMargin = % 16 with formula (title#:PARA.LeftMargin# + 6)%
--7--          } with tag list (PARA$) and
--8--                   attribute list (PARA.LeftMargin = 16
--9--                                   PARA.Raggedness = left
--10--                                  PARA.Justification = TRUE)
--11--      {formula: title#|
--12--       < Here is the title> } with
--13--                  tag list (PARA$)  and
--14--                  attribute list (PARA.LeftMargin = 10
--15--                                  PARA.Raggedness = centered
--16--                                  PARA.Justification = FALSE)
--17--      {formula: text#|
--18--      PARA.LeftMargin = % 16 with formula (title#:PARA.LeftMargin# + 6)%
--19--       < The first paragraph starts here> } with
--20--                  tag list (PARA$)  and
--21--                  attribute list (PARA.LeftMargin = 16
--22--                                  PARA.Raggedness = left
--23--                                  PARA.Justification = TRUE)  }
```

If the left margin of the title paragraph is changed by an editing operation, the left margin for the text paragraph will also be changed. If the left margin of the text paragraph were defined by a simple expression, changing the left margin of the title paragraph would NOT affect the left margin of the text paragraph.

Rule 17, 18

```
quotedExpression   ::=   """ expressionList """
expressionList     ::=   expression | expressionList expression
```

Quoted expressions are sequences of expressions delimited by double quotes. They appear as values for bindings. They are evaluated when the corresponding reference appears, not when they are first encountered. They are used to define expressions that need to be evaluated in an environment different than the one in which they are occur. Quoted expressions may only appear in bindings.

4

Base language: semantics

4.1 Internalization, evaluation, and externalization

The semantics provide the meaning of a script and state what a computer system should do with that script.

Unless altered by a user, a computer system should maintain in a script it externalizes what was found when that script was internalized. In the simplest case where a system does not change anything in a script, it should externalize a script equivalent to the one it internalized. But equivalence does not mean identity. Interscript, like any language, permits different phrases to mean the same thing even though different words are used. To externalize scripts equivalent to those internalized, a system must rely on the Interscript semantics, which states what a script means.

The semantics of the Interscript language are formally expressed in the Standard. Here we describe the semantics informally through illustrative examples. This kind of description should help determine the meaning of a script, much as a dictionary provides understanding for the meaning of words. However, any informal semantics description may leave ambiguities in the understanding of concepts. Interscript system implementors must refer to the Standard to avoid such misunderstandings.

Scripts consist of nodes; nodes have contents, bindings, and tags. Care must be taken when scripts are internalized because those tags, bindings, and content will have to be externalized eventually. Interscript does not require systems to represent scripts in a specific way inside the memory of a computer (such design issues are left to implementors). There are rules, however, that should be respected if a system is to "understand" a script correctly. Those rules give hints about how internalization should be done.

The succession of grammar rules used when generating a script form a syntactic tree called the parse tree of that script. For example, an Interscript node is a list of items. Those items make up the first level in the parse tree of that node. The Interscript semantics tell the meaning of a parse tree. To define the meaning of a node, we say the node is evaluated and the result of this evaluation should be consistent with the Interscript semantics rules.

Interscript system implementors need not implement a system that evaluates nodes. But they must implement a system that behaves according to the evalution rules fixed by the semantics as described in this chapter.

4.2 Nodes: contents and bindings

No matter which data structure is used, a node has content, bindings, and tags. The content of a node is deduced from the items in the parse tree by precise rules. Here are the first semantics rules for Interscript:

Rule 1: All items in the parse tree should be evaluated in a depth-first manner, from left to right and moving down when embedded nodes are encountered.

Rule 2: At any time the set of items that have been evaluated at that time form what is called an environment.

Rule 3: The content of a node is the sequence obtained when all items in the node that are expressions, formulas, and references have been evaluated.

Rule 4: Basic objects evaluate to themselves.

Rule 5: The evaluation of a non-structural and structural binding results in the association of the binding name with the corresponding expression. This binding is added to the environment.

Rule 6: The evaluation of a reference, when the entity referenced is a name bound to a value, results in that value.

Consider a node consisting of just terms:

```
{3   TimesRoman}
```

The contents of this node are the number 3 and the atom TimesRoman. Consider this node with four items:

```
{x=5  x#  x#  x#}
```

The first item binds the name x to the value 5; the next three terms are references to x. Therefore the contents of this node is the sequence of numbers 5 5 5. The binding is not part of the contents, but it is part of the environment until the end of the node is reached.

Non-structural bindings are used only to save bits in the machine encoding and to make things clearer in the publication encoding. They *need not be remembered* for the purpose of externalizing a node. {5 5 5} is an equivalent node to {x=5 x# x# x#}: a system might externalize either of the two nodes after internalizing the other. Thus two different scripts may be equivalent though they consist of different sequences of byte values.

Example 4.1 Node contents.

```
--1--    { y = 9 {4  y# 4} y = 8   5  y# 5 }
```

The first item in Example 4.1 binds y to 9. The second item is a node that must be evaluated next. This embedded node has three items. Since y is bound to 9, the contents of the embedded node is 4 9 4. The next item in sequence binds y to 8. The outer node

contents therefore consists of four values: a node containing 4 9 4, and the numbers 5, 8, and 5.

Here is a possibly *erroneous* node:

```
{ {y = 4  y# + 6 } z = 5 + y# }
```

The first embedded node is correct: its content yields the value 10. Assuming that no outer node contains a binding to y, the reference to y following this node is an error since non-structural bindings are only available within the node in which they appear. Thus the addition of 5 to the value bound to y violates Interscript semantics. (It would not be an error if y were defined in a node embracing this one.)

> **Rule 7**: Structural bindings are attached to the node in which they appear and must be preserved when that node is externalized.

The only difference between structural and non-structural bindings is that structural bindings must be remembered together with the node in which they appear. The node on line 1 in Example 4.2 may be externalized as it is or as the node on line 2. In any case, a structural binding to the variable x must appear.

Example 4.2 Structural binding.

```
--1--   { x = % 10  x = % 20  x# + 10 }
--2--   { x = % 20  30 }
```

> **Rule 8**: When the name used in a binding is a qualified name, a "record" named with the first identifer is searched in the present environment. If it does not exist, it is created. Then, the second name is searched within that record. If it does not exist, it is created. Then the value is bound to the name.

Example 4.3 Qualified names.

```
--1--   {  POINT.X = 10
--2--      POINT.Y = 20
--3--      POINT.X# + POINT.Y# }
```

In Example 4.3 POINT acts somewhat like a Pascal record with two fields, X and Y. This node contains the value 30 obtained by adding the values bound to POINT.X and POINT.Y. It might be externalized as the node: {30}. The dot naming convention is just a convenience. The node would be the same if the variables were simply called X and Y.

> **Rule 9**: When the first name of a qualified name is the name of a tag that exists in the environment, it is said to be a relevant binding. Relevant bindings must be externalized within the node in which they occur.

Example 4.4 Relevant bindings.

```
--1--   { FONT$
--2--      FONT.Points = 20
--3--      FONT.Boldness = regular }
```

Example 4.4 must be externalized as it is.

> **Rule 10**: A path is formed with references and qualified names separated by a colon. The evaluation of paths proceeds from left to right. The evaluation of the reference must yield a node. This node in turn should have a binding for the qualified name from which the evaluation proceeds.

Example 4.5 Paths.

```
--1--   {myline = { LINE$
--2--                LINE.AboveBaseline = {MEASURE$ MEASURE.Nominal = 10} }
--3--   linespace = myline#:LINE.AboveBaseline#:MEASURE.Nominal# * 2}
```

Example 4.5 shows a relevant binding that sets the value of AboveBaseline to a node that is a MEASURE node. This node contains a relevant binding to the Nominal attribute. On line 3 the linespacing is computed by accessing this value through its path. Finally the name linespace gets bound to 20. The reference of the path causes an inspection of the LINE node from which the AboveBaseline value is taken. Then an inspection of the MEASURE node yields the value 10 for the Nominal relevant binding. Note that a path can be used only in a reference. It may never appear on the left side of a binding. Values in inner nodes are *read only*; they cannot be modified from the outside.

4.3 Quoted expressions and formulas

Sometimes it is necessary to remember not only a value, but how that value was obtained. Assume the length of a line of text is to be 60 characters long. (Since Interscript's basic measurement unit is the mica [0.01 millimeters], in the following example each "character" is 25 micas wide.) This is NOT the way to set the right margin:

```
{  leftMargin  = % 10*25
   lineLength  = % 60*25
   rightMargin = % leftMargin# + lineLength# }
```

This is wrong because it does not achieve the desired goal. The node might be correctly externalized as :

```
{  leftMargin  = % 250
   lineLength  = % 1500
   rightMargin = % 1750}
```

Now if this node were internalized into another sytem, a user could reset leftMargin or lineLength without affecting rightMargin. Although that gives a correct Interscript node, the original intention of setting the rightMargin based on a computation involving leftMargin and lineLength would be lost.

For this example, it is not sufficient to perform the computation; the system must also remember that a formula was used to obtain the value of rightMargin. A formula (marked by a % symbol) is an expression that is remembered with the binding name. Example 4.6 shows the correct way to set the right margin.

Example 4.6: Formulas.

```
--1--   {  leftMargin  = % 10*25
--2--      lineLength  = % 60*25
--3--      rightMargin = % ( leftMargin# + lineLength# ) % }
```

Rule 11: The evaluation of a formula results in a value associated with the binding name. The difference with simple expressions is that formulas must be externalized as they were internalized. This explains why formulas can be used only with names from structural bindings.

Rule 12: The evaluation of a quoted expression results in the expression obtained by removing the quotes. More formally, it results in the parse tree obtained when parsing the expression, without evaluating this parse tree.

Rule 13: The evaluation of a reference where the entity referenced is a name bound to a quoted expression results in the evaluation of that parse tree in the environment of the reference.

Example 4.7: A simple document with a quoted node.

```
--1--   INTERSCRIPT/1.0
--2--   { DOCUMENT$
--3--      footer  = "{ CHARS$ PARA$ <Interscript -- September 1985>  pageNumber# }"
--4--      {PARA$
--5--         pageNumber = 1
--6--         {CHAR$ <This text is printed on the page>}
--7--         footer# }
--8--   }ENDSCRIPT
```

Line 3 in Example 4.7 binds the name footer to a quoted node. Because no evaluation is performed when the expression is encountered, there is no error even though pageNumber has an unknown value. When the footer reference on line 7 is evaluated, the content within the quote marks is substituted and evaluated. Then pageNumber is referenced, yielding the value 1.

4.4 Referencing nodes

The node reference facility is provided for the purpose of clustering common use information into a single node that is later referenced. It promotes script compactness and avoids tedious and possibly erroneous repetitions in a script. A node reference is like transferring a liquid from one container into another or like the include statement of many programming languages. The content of the identified node flows into the node where it is referenced and joins whatever is there. It is different from a quoted expression because content is evaluated at the time the node is bound. Consider the reference to x in Example 4.8 (line 1). The first # following x obtains the node containing ten digits; the second # removes the braces. The content of the outer node is the digits from 0 to 9 repeated three times. A less compact way to externalize this node is shown on line 2.

Example 4.8: Referencing nodes.

```
--1--   {x = {0 1 2 3 4 5 6 7 8 9}  x# #  x# #  x# #}
--2--   {0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9}
```

A collection of bindings may also be grouped into a node (e.g. a collection of default values). Such bindings must be structural (or relevant) or they will not be retained by the node evaluation. Those bindings may simply be recalled elsewhere by node references as shown in Example 4.9.

Example 4.9: Referencing nodes to introduce defaults.

```
--1--   INTERSCRIPT/1.0
--2--   { DOCUMENT$
--3--       title = % {PARA$
--4--                   PARA.LeftMargin = 10
--5--                   PARA.Raggedness = centered
--6--                   PARA.Justification = FALSE}
--7--       {                                    -- chapter 1 starts here--
--8--                   {title# #  <Title of chapter 1>} }
--9--       {                                    -- chapter 2 starts here--
--10--                  {title# #  <Title of chapter 2>} }
--11--  }ENDSCRIPT
```

Although this is correct, Example 4.9 could produce Example 4.10 during externalization. We say, therefore, that 4.9 and 4.10 are equivalent. However, a user's intent will be lost if the script shown in Example 4.10 is produced.

Example 4.10: Equivalent script to Example 4.9.

```
--1--   INTERSCRIPT/1.0
--2--   {DOCUMENT$
--3--       title = % {PARA$
--4--                   PARA.LeftMargin = 10
--5--                   PARA.Raggedness = centered
--6--                   PARA.Justification = FALSE}
--7--       {                                    -- chapter 1 starts here--
--8--                   {PARA$
--9--                    PARA.LeftMargin = 10
--10--                   PARA.Raggedness = centered
--11--                   PARA.Justification = FALSE
--12--                   <Title of chapter 1>} }
--13--      {                                        -- chapter 2 starts here--
--14--                  {PARA$
--15-                    PARA.LeftMargin = 10
--16--                   PARA.Raggedness = centered
--17--                   PARA.Justification = FALSE
--18--                       <Title of chapter 2>} }
--19--  }ENDSCRIPT
```

There is no indication in this script that a chapter title is influenced by the characteristics of titles in general (lines 3-6). If a generalized title's LeftMargin is changed to 20 (see line 4), it

would not influence the margin for the titles of chapters because there is no indication these values (on lines 9 and 17) were obtained by referencing a node bound to title. To maintain that relationship, the node reference must be accompanied by a formula as shown in Example 4.11.

Example 4.11: Using a formula to remember the references.

```
--1--   INTERSCRIPT/1.0
--2--   { DOCUMENT$
--3--       title = % {PARA$
--4--                   PARA.LeftMargin = 10
--5--                   PARA.Raggedness = centered
--6--                   PARA.Justification = FALSE}
--7--       {                                    -- chapter 1 starts here--
--8--                   { title# #% <Title of chapter 1>} }
--9--       {                                    -- chapter 2 starts here--
--10--                  { title # #% <Title of chapter 2>} }
--11--  }ENDSCRIPT
```

4.5 Node construction

Tags have attributes and nodes may carry tags. Whenever a node has some tag, it may specify the attribute values of that tag by simply binding the attribute name to the desired value. Such bindings are called the relevant bindings for that node. To maintain the base language syntax, relevant bindings take the same syntactic form as non-structural bindings (name = term). This will not be the case in the machine encoding where non-structural bindings, relevant bindings, and structural bindings will be distinguished.

There will be no ambiguity in the machine encoding between an attribute name and a non-structural binding name. For publication encoding, it is good practice to specify all the tags on a node first, followed by the relevant bindings. They may then be understood immediately.

Here is the semantics rule concerning the construction of a node:

Rule 14: To construct a node, evaluate all items in the node and then:

- —Get the content items.
- —Retain the tags and their attribute values that are assigned using relevant bindings.
- —Retain the structural bindings attached to the node, if any.

5

Safety rules for Interscript systems

Interscript makes it possible for systems to manipulate the parts of documents they understand without altering parts they do not. The second section of this chapter shows how a system may safely process nodes even if that system does not specifically implement the tags carried by those nodes.

5.1 Standard tags and their implementation

A system might be based on the evaluation rules enumerated in the semantics. It could build specific memory constructs called items, nodes, tags, and so on. A data structure containing those constructs would be obtained through an evaluation mechanism respecting the enumerated rules. Such a system is said to interpret layer 2 through layer 1. This is because there is nothing like a *page* or a *paragraph* in such a system. It only knows about nodes and tags. The definition of each tag is kept as a specific node in some sort of symbol table. For each binding, it would be necessary to look in the symbol table to know if this binding is relevant and find the attribute definition to check its consistency with the value. This definition in turn might be a new node that must be interpreted.

Such a system can be interesting for developing experimental software. For the purpose of error handling in erroneous scripts, it would say exactly why and where a script is erroneous. It may be useful too to check the output of another system. But it would consume excessive memory space and operate slowly because it would always act as an interpreter.

Consider now a system that wants to take advantage of the PARA$ tag. Although this tag has not yet been defined precisely, let us assume it specifies that sequences of characters should be printed with certain line spacing, within specified margins, and with or without line justification. Assume also the system is coded in Pascal. It may then define `Paragraph` as a `record` type and its contents as a list of strings. The attributes of the PARA$ tag are then declared as `fields` of this `record`, with their type matching the tag definition.

Our system does not know any memory construction called `node`. Thus, whenever this system internalizes a PARA node, it does not create a node structure in memory. It does create a new instance of a `paragraph` record. Since PARA$ is a predefined tag, its attributes are known. The system does not need to lookup attribute names for PARA$ in some symbol table. It knows what they are and simply fills in the record fields with the relevant binding values it finds in the script. For the fields that have not been assigned,

a simple procedure may supply default values that are also known from the PARA$ tag and may be represented as Pascal constants in the program. The paragraph contents may be obtained by building a list with all successive strings found in the node with their different fonts.

Fast systems result if some tags have been predefined. If PARA$ were not pre-defined, we could not encode a program that knows about paragraphs. This is why the Interscript Standard defines a particular set of tags called the standard tags. Those tags are either tags used in conjunction with the imaging model developed in chapter 7, or tags that are known to be of general utility. Of course the Standard may be extended by adding new standard tags from time to time. The machine encoding also provides a more compact encoding for standard tags than for non-standard ones.

Systems that embed knowledge of standard tags in their code are said to implement tags. Most systems will implement all the standard tags (that is why standard tags have been defined) in the fastest possible way.

5.2 Unknown tags

A particular organization or software manufacturer might also develop its own systems with a set of its own private tags. Such systems will be able to interchange scripts correctly, while taking advantage of more sophisticated features.

But this raises a question for the systems that do not implement these private tags: What do they do if they receive such scripts? The question also applies to systems that do not implement all standard nodes. They receive scripts they do not understand entirely, that is, they cannot interpret the meaning of the nodes according to the tag. Since most tags concern layout, a system receiving unknown nodes cannot display these nodes as intended. To handle a node correctly a system must know what its tags mean. Some operations, however, can be performed on nodes with unknown tags.

Consider a system that does not implement the standard tag named INKED$. This tag describes a rectangular area totally filled with some colored ink. A system receiving an INKED node may know nothing about this tag. It cannot display the inked rectangle and it cannot do anything to the INKED node. Another system might know that INKED$ is a tag that has a color attribute. Even though it is not able to display the inked rectangle, it may allow the editing of the color attribute.

Thus there are several possibilities for dealing with non-understood nodes according to a system's capabilities.

5.3 Editing totally unknown nodes

Consider a system that implements only a subset of the standard tags. Such a system would do nothing for other tags and would not know the definition of the standard tags it does not implement. However this system conforms to Interscript and knows the syntax for nodes and tags.

Assume this system implements the CHAR node and that it receives the node in Example 5.1 with color a relevant attribute for UNKNOWN$.

Example 5.1 Node with UNKNOWN$ tag containing immediate and indirect contents.

```
--1--   {UNKNOWN$ UNKNOWN.Color = <black>
--2--    {CHARS$ <the cat is > } {CHARS$ UNKNOWN.Color| } }
```

This system cannot display the node as intended, but can manage this node in either of two ways:

- The system does nothing with these nodes because it does not implement everything. The nodes must be saved somewhere. They must not be modified and should be externalized as received. This allows the interchange among systems with different capabilities. One wouldn't expect a spreadsheet system, for example, to implement the display of complex graphics.

- The system might display the nodes to a user (with a warning) to allow some editing since it knows how to display the inner nodes. In this example a user might change the word <cat> to <parrot>.

We can prove from Interscript semantics that immediate values in the *node contents (*of this example*)* may be replaced by another value *with the same tag.*

In the case of indirect content the values in the nodes' content that were obtained through simple references may be changed. Example 5.1 is semantically equivalent to the node in Example 5.2. Hence the value in the second CHARS node may be changed also.

Example 5.2 Node with UNKNOWN$ tag containing nodes with immediate contents.

```
--1--   {UNKNOWN$ UNKNOWN.Color = <black>
--2--    {CHARS$ <the cat is > } {CHARS$ <black>} }
```

Consider the question of modifying the bindings. A system cannot do anything to the relevant bindings of an unknown tag. It does not know, for example, what values are allowed for the Color attribute. A system that assumes "since the value is a string, it can be replaced by a string" would be wrong.

Example 5.3 Attribute of UNKNOWN$ tag referenced in a formula.

```
--1--   {UNKNOWN$ UNKNOWN.Color = <black>
--2--    {CHARS$ <the cat is > } {CHARS$ UNKNOWN.Color|% } }
```

In Example 5.3 the content of the second CHARS node is obtained by a formula. In this case the script clearly specifies that the content comes from the binding and should be kept identical. Consequently it is not possible to modify the content. This node must be externalized as it is and the value of the binding cannot be changed.

We may now summarize the two possible actions a system might take when it receives a node with an unknown tag:

1. It may not modify any binding in the node nor modify the contents obtained by formulas, or
2. It may modify the values in the contents if their tags are maintained.

6

Document architecture

The preceding chapters discussed layer 1 of Interscript, called the base language. Information expressed in this base language may be interchanged and will be *formally* understood by the recipient, i.e. the recipient can identify the constituents like nodes, tags, bindings, etc., and transform them into an internal structure (this is the internalizing process). But the base language is not able to express the meaning of the content. In particular, the base language has no constructs specifically for a document; it only describes scripts and knows nothing about documents.

The next chapters discuss layer 2 of Interscript. They define the standard document constructs and describe how to use those constructs for expressing documents as scripts. This chapter presents the *philosophy* underlying those constructs. It gives an introductory overview of the document architecture and interchange formats used in Interscript.

6.1 Architectural concepts

What a user perceives as a document is a conceptual unit with varying aspects and is viewed by different users in different ways. Before we standardize document interchange, we need to specify that conceptual unit. The document architecture clarifies that concept by introducing the constructs and rules from which documents are built.

6.1.1 What is a document

"Document" is a fuzzy concept. The *information* that a document contains or conveys can vary depending both on the nature of the document itself and the beholder. Telegrams, a printed memo, and an illuminated manuscript differ considerably. A book on physics is viewed differently by a professor, a student, and a typographer.

Recent efforts (especially within ECMA, CCITT, and Xerox) to standardize document interchange have brought a better and more formal understanding of documents. ECMA, in its Office Document Architecture (ODA) [Reference 7], and CCITT, in its document interchange protocol for the telematic services [Reference 8], define documents this way:

> A document is a structural amount of text that can be interchanged as a unit between an originator and a recepient.
>
> Text is information for human comprehension that is intended for presentation in a two-dimensional form, e.g. printed on paper or displayed on a screen. Text consists

of graphic elements such as character box elements, geometric elements, photographic elements, and combinations of these.

The content of a document consists of text and some additional control information.

6.1.2 Constituent parts of a document

Essentially a document consists of these parts:

- Profile
- Logical definitions
- Layout definitions
- Logical structure
- Layout directives
- Layout structure
- Common content
- Content

Documents need not contain all these parts. But those parts present in a document must be captured by an editor and faithfully stored and transmitted.

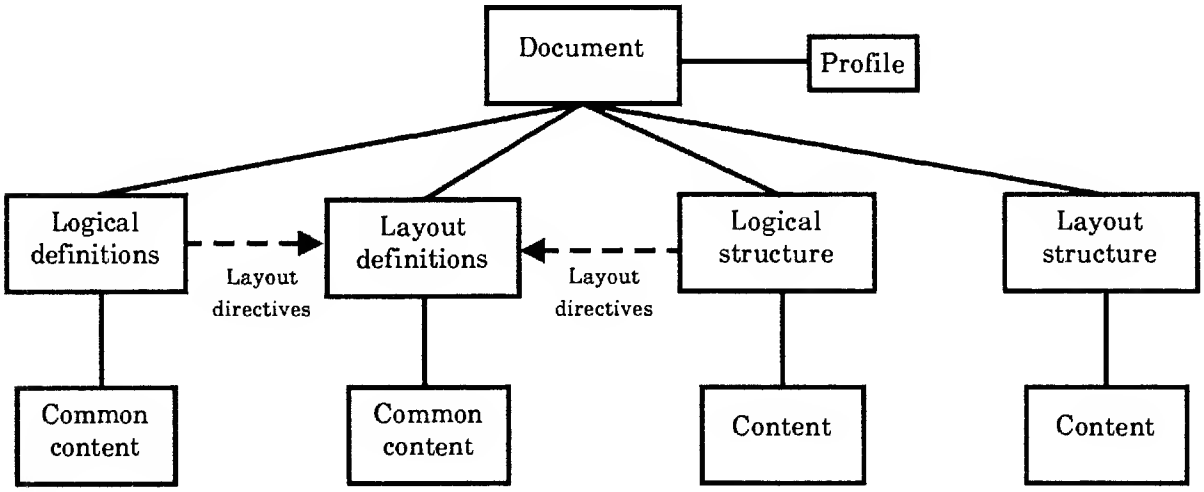

Figure 6.1: Constituent parts of a document

6.1.3 Layout structure

An "ordinary" user's view of a document is as one or more pages containing character text and illustrations. A user sees the character text assembled in lines that fill one or more columns along with illustrations contained in boxes. Lines, columns, boxes, pages, etc., are layout objects that are nested (pages contain columns and these, in turn, contain lines) and form the layout structure of the document.

When considering a document from the layout point of view, the content appears divided into content portions fitting into the "leaf" layout objects of the layout structure (such as Lines in Figure 6.2).

6.1.4 Logical structure

A document is not the black and white pattern of ink on a page; a user "reads more into it." The user attaches significance to the text characters that comprise the words in the document, rather than to the exact pattern of ink or toner that display the letter-shapes. A user attaches significance to some of the typographic characteristics of the document: words may be emphasized by font selection, lists may be constructed by indenting. Supported by the spatial arrangement, a user recognizes other logical objects such as paragraphs and chapters. All those logical objects are nested (chapters contain para-

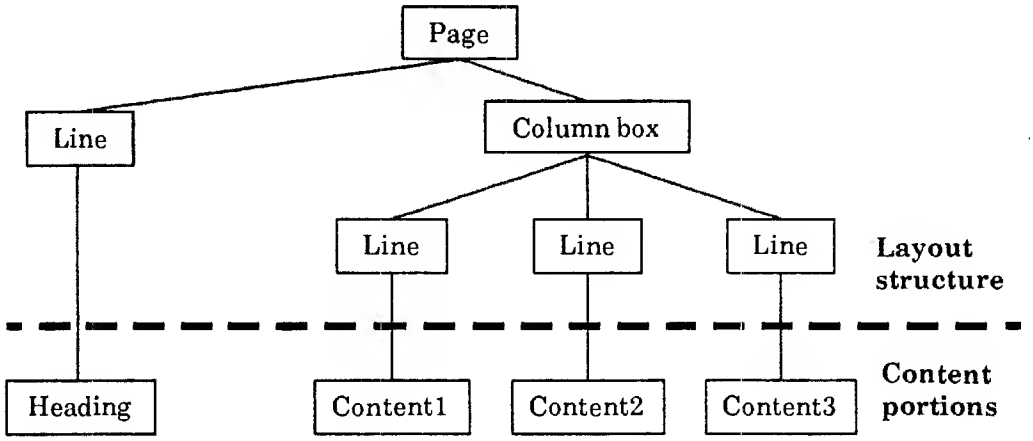

Figure 6.2: Layout structure with content portions

graphs and these, in turn, contain emphasized words), forming the logical structure of the document.

When considering a document from the logical point of view, the content appears divided into content portions belonging to the "leaf" logical objects of the logical structure (such as Paragraphs in Figure 6.3).

6.1.5 Layout and logical definitions

When the content of a document is filled into the layout structure, the resulting content portions are adapted to the containing boxes and are therefore difficult to edit. For example, inserting a character into a line may cause the last character in the line to extend beyond the line boundary. The layout structure does not specify what should be done in that case. It does NOT contain any information about how it is to be changed. Such information is provided by layout definitions.

Layout definitions contain rules that define how to create and change layout structures. They may determine whether to squeeze the characters in a line such that an inserted character fits, or to create a new line. They may define special page layouts (e.g. the

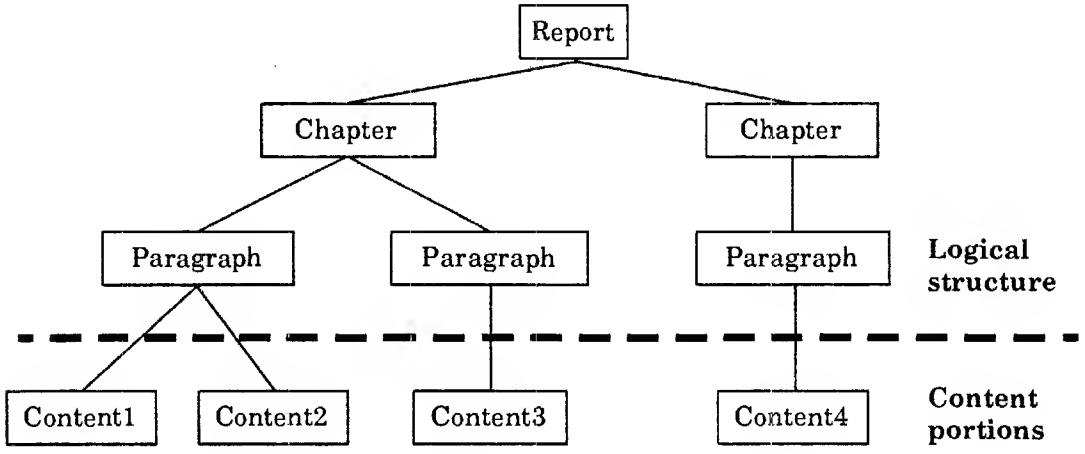

Figure 6.3: Logical structure with content portions

partitioning of a page into a fixed heading box, a "stretchy" content box, and a fixed footing box) or special chapter layouts (e.g. how a chapter title should be rendered).

Logical definitions contain rules that define how to create and change logical structures. The logical definition of a chapter may require, for example, that each chapter consists of a title and an arbitrary number of a special paragraphs.

Layout definitions and logical definitions may contain some fixed content portions that appear in each instance of the definition. For example, a user may put some fixed text into a footing box of a page definition that is rendered each time a page is created according to that definition. Such content portions are called common content.

For rendering logical objects a layout definition is necessary. Layout directives provide this information. They may be attached directly to logical objects ("this object must be laid out according to that layout definition") or to logical definitions ("each instance of this logical definition must be laid out according to that layout definition").

6.1.6 Profile

It is often useful to attach information to a document that facilitates the handling of the document as a whole. Such information is carried with a document but not rendered. The user may want to associate an author's name or the creation date with a document. Such information is carried in the profile.

6.2 Presentation and representation of documents

Within the electronic office, what a user *sees* as a document and what is *stored electronically* as the document are two different things. The former shows certain aspects of the document depending on the output device (this is the *presentation*). The latter is a machine-readable dataset containing all relevant information, and can be precisely defined (this is the *representation*).

6.2.1 Representation of documents

An editor must operate on a well-defined machine-readable form of a document. As discussed in section 3.2, the internalizing and externalizing process produce such forms: when internalized, a document is transformed into its internal representation; when externalized, it is transformed into the machine encoding. We call those machine-readable forms a representation of a document.

Something that cannot be expressed in an editor's representation cannot be included in a document that editor manipulates. Thus the form of the representation controls, to some extent, the power of an editor.

Interscript is one such representation. Since it is designed to be general-purpose, it is important that it be able to represent arbitrary document content.

The problem of taking a user-perceived piece of document content and capturing it within an editor's representation is the main task of this standard.

6.2.2 Presentation of documents

How an editor *shows* a document to a user is greatly affected by the nature of the machine on which the editor runs. Some editors have primitive display mechanisms, some are more complex. Some editors may output a document to a CRT and to an attached printer. These are all presentations.

Often, an editor is prepared to present a document on a printer in a form a user thinks of as its "finished" form. This is called rendering the document. Rendering may be a capability of an editor, or it may be an "offline" activity performed by a separate software package.

The presentation of a document so a user can get feedback during an editing session is called display. A display usually only shows a fragment of a document, and frequently the display is in a form quite different from the "finished" form that will be rendered.

Some editors attempt to display a document in the same form as it will appear when rendered. Such editors are called WYSIWYG ("what you see is what you get") editors; they typically use a bitmap-style raster output device to achieve reasonable fidelity to the as-rendered image. Since such editors must, in effect, render a document on-line and in real-time, they require considerable computing power to achieve good performance when displaying complex documents.

Other editors display a document by showing the user a stylized (and not final-form) version of the document—the presence of italic text may be indicated by the appearance of `/italic/` on the display. This is often the case with editors that operate on character-CRT workstations.

6.3 Interchange formats

The interchange format that should be chosen for a document depends on a sender's expectation of what a recipient will do or should be allowed to do with the document. Interscript offers an *alternative*: a sender may transmit a document either in *image format* or in *processible format*. Both formats use a common *content format*.

6.3.1 Content format

At present Interscript supports only character content; future versions will contain content formats for graphics, images, spreadsheets, and other content types.

In principle, Interscript decomposes each character content portion into a sequence of individual characters and defines an individual character as a CHAR node.

Interscript does not require one particular character set; a document originator may choose any character. The name of the character set is put into the CHAR node by binding it to a special CHAR attribute.

Control characters (i.e. characters that affect the processing or interpretation of a content portion without being rendered such as "New Line") may be contained in a character set. Control characters may also be provided by special Interscript tags (e.g. the TAB$ tag defining tabs).

Characters are rendered in character boxes; the necessary information about the box dimensions is given by the FONT$ tag. Characters serve two functions: from the logical viewpoint they represent the logical content; from the layout viewpoint they are small boxes containing information about a certain distribution of ink within the box area.

Section 7.3 will discuss content formats in detail.

Example 6.1: A character content portion of three characters (elides detail).

```
--1--   {CHAR$ FONT$
--2--          CHAR.Characterset = XeroxCharacterSet
--3--          FONT.Name = Helvetica
--4--          <T>}
--5--   {CHAR$ FONT$
--6--          CHAR.Characterset = XeroxCharacterSet
--7--          FONT.Name = Helvetica
--8--          <h>}
--9--   {CHAR$ FONT$
--10--         CHAR.Characterset = XeroxCharacterSet
--11--         FONT.Name = Helvetica
--12--         <e>}
```

6.3.2 Image format

When a recipient is expected only to render the document but not to edit it, the image format (also called final format) is used. It describes the rendered document comprising the *Layout structure* and the *Content* possibly with the *Profile*.

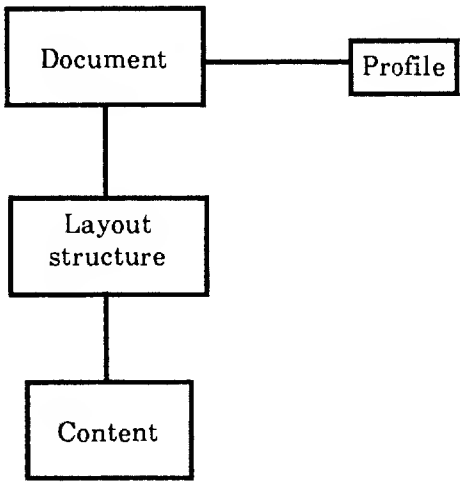

Figure 6.4: Image format of a document

Interscript knows only one type of layout object: the box defined by the BOX node. A box is a rectangular area on the page with its edges parallel to the page edges. Lines and pages are defined as special boxes by the LINE$ and PAGE$ tag, respectively. All boxes in the image format have a *fixed size*. When nested, they have a distance measure in the x- and y-direction to the containing box that fixes their relative position.

Boxes are not specific to the image format; they appear in the processible format where many more of their features are used.

In Interscript image format documents are scripts whose dominant hierarchy represents the layout structure. This layout structure is a sequence of page boxes each of which contains a nested tree of subboxes.

The content in an image format document is formatted (i.e. divided into content portions and filled into the leaf boxes of the layout structure). Since image format documents represent the result of rendering, the content is not expected to be edited or reformatted.

Example 6.2: The layout structure of Figure 6.2 expressed in Interscript (elides detail).

```
--1--   {BOX$ PAGE$                                          Page box
--2--       {LINE$                                           Heading line
--3--           {CHAR$ <..>} ... {CHAR$ <..>}}               Sequence of characters
--4--       {BOX$                                            Column box
--5--           {LINE$
--6--               {CHAR$ <..>} ... {CHAR$ <..>}}           Content1
--7--           {LINE$
--8--               {CHAR$ <..>} ... {CHAR$ <..>}}           Content2
--9--           {LINE$
--10--              {CHAR$ <..>} ... {CHAR$ <..>}}}}         Content3
```

The image format of a document must not be confused with the Interpress master of the document. Both formats represent the rendered document in a device-independent form; but the former is a structural description as perceived by people, and the latter consists of procedural instructions for a "vitual" printer. Transforming image format documents into Interpress masters is often expensive in terms of computing time; this task must be done by a separate printing procedure.

6.3.3 Processible format

When a recipient is supposed to edit and reformat a document, the processible format (also called revisable format) is used. This format allows several users on different systems to interchange and edit the same document. Compared to the image format, it is much more complex. Establishing such a format is an ambitious task and is the *main goal* of Interscript.

The processible format separates the content from the layout. It comprises the *Logical structure* and the *Content* of the document, *Layout definitions* attached to the objects of the logical structure possibly with *Common content* portions, and possibly the *Profile*. (See Figure 6.5.)

Note that this version of Interscript is not capable of expressing logical definitions. It does not allow characterizing special logical objects as instances of, say, a chapter definition. But it does allow attaching a layout definition to this special logical object such that the rendered object looks like a chapter.

Note also that this version of Interscript does not allow processible documents to contain their layout structure. Each recipient of a processible document must convert it into displayable form.

Interscript knows only two special types of logical objects: documents and paragraphs as contained in the DOCUMENT and PARA node, respectively. All other logical objects are given as ordinary nodes. Logical objects to be rendered obtain their necessary layout directive from the POUR$ tag whose Template attribute is bound to a layout definition. PARA nodes are implicitly POUR nodes with a predefined layout definition.

In Interscript a processible document is a script whose dominant hierarchy represents the logical structure. This logical structure is a nested tree of nodes rooted in a DOCUMENT node.

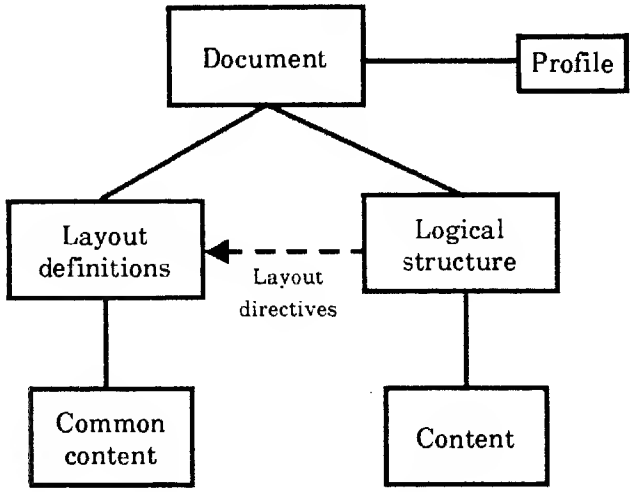

Figure 6.5: Processable format of a document

The content is divided into content portions according to the logical objects to which they belong. In contrast to the image format, the content portions are here *separated from the layout*. This allows them to be edited.

In Example 6.3 we assume that the global environment contains bindings for ReportLayout and ChapterLayout providing layout definitions for "reports" and "chapters."

Example 6.3: The logical structure of Figure 6.3 expressed in Interscript (elides detail).

```
--1--   {DOCUMENT$ POUR$                                          A report
--2--       POUR.Template = ReportLayout|
--3--       {POUR$                                                First chapter
--4--           POUR.Template = ChapterLayout|
--5--           {PARA$
--6--               {CHAR$ <..>} ... {CHAR$ <..>}                 Content 1
--7--               {CHAR$ <..>} ... {CHAR$ <..>}}                Content 2
--8--           {PARA$
--9--               {CHAR$ <..>} ... {CHAR$ <..>}}}               Content 3
--10--      {POUR$                                                Second chapter
--11--          POUR.Template = ChapterLayout|
--12--          {PARA$
--13--              {CHAR$ <..>} ... {CHAR$ <..>}}}}              Content 4
```

In contrast to the layout structure, layout definitions must be capable of handling content portions of *varying sizes*. For example, they must require the construction of a new page when the content "flows over" the previous page, or allow boxes to disappear when no content is received. Providing such rules is the central task of layout definitions.

Interscript solves this problem by providing two constructs:

- **Templates**
  Layout definitions are given as regular box expressions built with TEMPLATE nodes. These expressions enable the user:
  - — to require that certain boxes must appear in the layout (e.g. a page must contain a heading box, content box, and footing box). These are sequences.

— to allow that a certain box be repeated arbitrarily often (e.g. a page box according to the content that is filled in). These are repetitions.

— to offer a choice between certain boxes (e.g. between one-column and two-column pages). These are alternations.

- **"Stretchy" boxes**
  Boxes appearing in layout definitions are "stretchy." Their size measures and distances are determined according to their content and the surrounding boxes. In layout definitions, boxes that should disappear when no content is received have "synthesized" size measures: when such a box is constructed for the layout structure, its size is calculated such that the content just fits.

Example 6.4 illustrates how a simple page would be defined in a template in Interscript.

Example 6.4: The layout definition of a simple page (elides detail).

```
--1--   {BOX$ PAGE$ TEMPLATE$
--2--       TEMPLATE.expresses = sequence
--3--       {BOX$ .. some fixed measures .. }          Fixed heading box
--4--       {BOX$ .. some synthesized measures         Synthesized content box
--5--           {TEMPLATE$                               containing
--6--               TEMPLATE.expresses = repetition      arbitrarily many
--7--               {LINE$ }}                            lines
--8--       {BOX$ .. some fixed measures .. }}         Fixed footing box
```

The layout definition constructs are explained in detail in section 7.5.

6.4 Formatting process

Since layout and content are separated in the processible format, they must be combined when the document is rendered. Boxes must be created according to the layout definitions; content must be divided in appropriate portions fitting into the boxes; the resulting boxes must be arranged onto the presentation medium. This is done by the formatting process.

The principles of the formatting process come from the hot-metal days of typography. At that time "dummies" were used to provide the page layout and "galleys" to provide the running text.

A dummy divided a page into boxes filling some of them with fixed content and leaving "holes" in the remainder. These holes had to be filled with text from galleys.

In Figure 6.6 there are four holes with three labels. For each label there must be a corresponding galley. The galley labelled with T will first be filled into the left column and the reminder into the bottom right box.

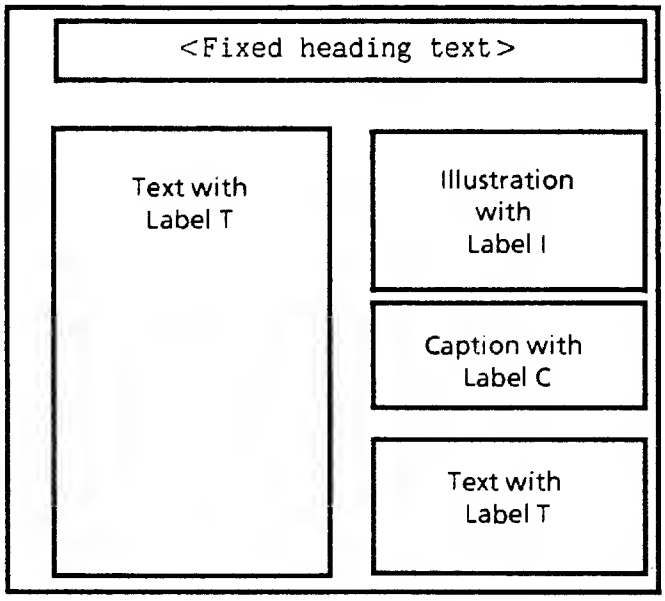

Figure 6.6: A dummy

Galleys containing text are type-set and broken into lines. They could be regarded as a series of lines on one long sheet of paper, to be cut and rearranged on pages.

Figure 6.7 shows three galleys that are to be placed, line after line, into the corresponding holes of the dummy in Figure 6.6.

Interscript's formatting concept is based on this interaction of dummies and galleys. In Interscript, the holes in dummies are called "MOLD boxes;" filling those holes is called "pouring."

Pouring is the main activity of the formatting process. Normally there are multiple levels of pouring: paragraph text (regarded as a sequence of character boxes) will first be poured into lines producing a sequence of lines. This sequence of lines will then be poured into a content box. Pouring will be explained in detail in Chapter 8.

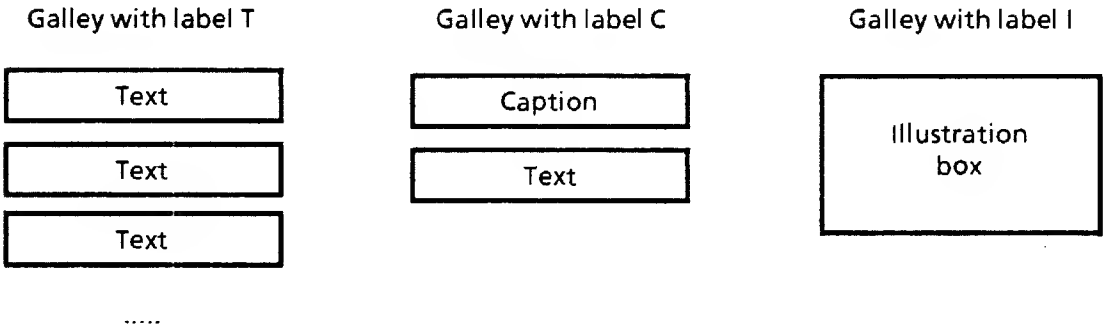

Figure 6.7: Three galleys meant to fit in the dummy of Figure 6.6

7

Standard document constructs

This chapter describes how documents can be expressed and interchanged by using the base language. For that purpose a set of standard document constructs forming Layer 2 of Interscript is created. This set builds up the processable and image format of documents determining the functionality of Interscript. Since Interscript is designed to be a general purpose standard for document interchange, the set of standard document constructs is intended to be as comprehensive as possible. This is certainly not the case for the present version. Future versions will standardize additional constructs.

Formally, the standard document constructs are given as tags in the base languge. Since a mechanism for defining tags is not available in the present Interscript version, the standard document constructs are informally presented by listing their attributes, sometimes with a Required Tags or Content property. The Required Tags and Content property are "typed;" the allowed "types" are contained in the list. Any attribute may be bound formally to the atom match regardless of its type (although match has meaning only for MOLD nodes). This will not explicitly be mentioned in the list. OneOf, SetOf, SequenceOf, and | (as a shorthand for ChoiceOf) are used to construct types. The types Any and Nil allow an arbitrary value and no value, respectively.

The standard document constructs are expected to be an integral part of most Interscript editors. Since Interscript is open-ended, *additional document constructs* may be created by an editor by using non-standardized tags; these constructs may be handled like standard ones.

7.1 Document as an entity

A document is defined as a script whose highest node carries the DOCUMENT$ tag. Thus a DOCUMENT node contains all the other nodes comprising a document. The attribute bindings of the DOCUMENT$ tag define a list of properties that concern the document as a whole and must be carried with the document. This list is the document's profile.

Tag: **DOCUMENT$**

Attributes

```
DOCUMENT.CreationTime: Integer          -- Use XNS standard
DOCUMENT.LastAlterationEditor: Atom     -- Editing software
                                           system that last changed the document.
DOCUMENT.LastAlterationTime: Integer    -- Use XNS standard
DOCUMENT.Title: CHAR node
DOCUMENT.Author: CHAR node
DOCUMENT.Owner: CHAR node
DOCUMENT.EditReason: CHAR node
```

Example 7.1: A document node.

```
--1--   {DOCUMENT$
--2--       DOCUMENT.Author = {CHARS$ <William C. Smith>}
--3--       DOCUMENT.EditReason = {CHARS$ <Update node definitions>}
```

The document node bindings contain information that describes the document itself. It does not contain information about how a document is to be handled by other applications; such information is outside the scope of Interscript.

Of course a document has many more characteristics when considered as a whole. These are provided by attributes of further tags that are added to the DOCUMENT node or to nodes contained within the DOCUMENT node.

7.2 Relations and labels

An essential part of a document is the relations between a document's components. These relations define rules for editing and formatting. For example, when a name appearing in two places in a document is changed, an editor might ask a user whether the new name should replace both instances or not.

The base language provides a straightforward method to express hierarchical relations by using the hierarchical node structure of a script together with the rules about scope and inheritance of bindings.

Not all relations can be expressed by such a hierarchical method. Interscript also provides a uniform way to describe non-hierarchical relations. Labels are used to attach some naming information to nodes in the logical or layout structure establishing relations between all nodes with matching labels. Labels are defined by tags; in contrast to the hierarchical method, non-hierarchical relations are evaluated only by an editing process.

All non-hierarchical relationships in a script must be expressed via the label facility of Interscript. There is no such thing as an "implicit" non-hierarchical connection. This enables any editor to discover all document connections in order to enable non-destructive editing.

7.2.1 Labels

Labels may be attached to nodes by using the LABEL$ tag.

Tag: **LABEL$**

Attributes

LABEL.Labels: {SetOf Atom}

The attribute Labels must be bound to a set of atoms that are called label-atoms. The phrase "a node is labelled with an atom" is a short way of saying "a node is given the LABEL$ tag and that atom is among the elements of the set bound to the Labels attribute." Labels are not user-sensible; they are supplied by an editor.

Labels express two forms of intra-document relations:

- **Pointers**
  These relations are "directed." One node is distinguished as the *source* node and contains a label used to point to another node, the *target* node. For example, a source node may reference the page number of a certain chapter (the target). Pointers are defined by certain tags with a from attribute that is bound to a label-atom (a typical example is the REMOTEBINDING$ tag of section 7.2.2). The source node is always the node where such a tag appears and the target node is defined by the label-atom bound to the from attribute.
- **Set and sequence relations**
  These relations are not "directed." A unique label-atom is chosen. Each node that is a member of the set or sequence is labelled with that atom. For a set relation, the order of its elements is not relevant. For a sequence relation, the order is dictated by the script order. Some editing processes invoked by other tags will evaluate such relations. A typical example is the set of nodes labelled for the pour operation (see section 7.5.1).

7.2.2 Remote tags

Remote tags may augment any node. They are used when information is to be derived from another node which cannot supply this information at the moment when the internalizing process encounters the remote tag. This information will later be provided and transferred by special editing operations typically occuring during a pour. This information may consist of the content of a node (indicated by the REMOTECONTENT$ tag) or the value of a relevant binding (indicated by the REMOTEBINDING$ tag).

Tag: **REMOTECONTENT$**

Attributes

REMOTECONTENT.FromNode: Atom

Tag: **REMOTEBINDING$**

Attributes

REMOTEVALUE.FromNode: Atom
REMOTEVALUE.FromBinding: Atom
REMOTEVALUE.Value: Atom

The REMOTECONTENT$ tag and the REMOTEBINDING$ tag have a FromNode attribute that must be bound to a label-atom. This label-atom must point to a *uniquely* defined target node in the logical or layout structure .

In case of REMOTECONTENT$, an editing operation copies the content of the target node into the REMOTECONTENT node overwriting the old content if any.

In case of REMOTEBINDING$, the FromBinding and the Value attribute are bound to attribute names. The former names an attribute in the target node whose relevant binding is to be copied. The latter names the attribute in the REMOTEBINDING node that is to be bound to the copied value. An editing operation creates that desired binding overwriting the old value.

Example 7.2: A REMOTEBINDING node .

```
--1--   {DOCUMENT$
--2--       ... {REMOTEBINDINGS$  T1$
--3--             REMOTEBINDING.FromNode = L
--4--             REMOTEBINDING.FromBinding = T2.A2
--5--             REMOTEBINDING.Value = T1.A1
--6--             T1.A1 = 5 ... }
--7--       ... {T2$  LABEL$                          The target node
--8--             T2.A2 = 10                           uniquely defined by
--9--             LABEL = {L} ... }                    the label L
```

The editing operation looks for a node with label L, copies the value 10 of the attribute T2.A2 in that node, and changes the binding of T1.A1 in the REMOTEBINDING node to 10.

Of special importance are references to nodes, which provide some numbers associated with the node they point at. This is typically the case with a reference to an illustration, say, to get its page number. A reference is expressed as a REMOTEBINDING node where the FromNode attribute points to the referenced node and the FromBinding attribute points to the name of the desired numeration. When the referenced node is moved within a document, the reference doesn't change (but the value of the reference does). It is not necessary to put an "anchor" into the referenced node. TEXTSHOW nodes must be used to make the value appear in the appropriate format (see section 7.3.5).

7.2.3 User-sensible naming

It is often convenient for an editor to attach a user-sensible name or description to a node. This could be used by an editor to prompt a user for some data-value. Interscript provides the USERNAME$ and the USERDESCRIPTION$ tags to augment any node.

Tag: **USERNAME$**

Attributes

USERNAME.Name: CHAR node

Tag: **USERDESCRIPTION$**

Attributes

USERDESCRIPTION.Description: CHAR node

7.3 Content architecture

In this document, the word "content" is used with two different meanings. When speaking about the nodes of the base language, "content" refers to the content of a single node (i.e. everything that is between braces). When speaking about documents, "content" refers to the content of an entire document, typically what one sees when the document is printed. "Content portion" is a subdivision of content in the latter sense.

Content portions are always attached to the leaf nodes of the logical and/or layout structure and never carry an internal node structure. The set of structures used to describe a content portion is called the content architecture. There are many content architectures, each corresponding to a particular "type" of content. This version of Interscript defines a character content architecture. Future Interscript versions will define a content architecture for graphics, images, spreadsheets, and other content types.

7.3.1 Character nodes and character boxes

Characters are expressed in Interscript as CHAR nodes which may contain one or more characters. Each character is uniquely identified by its name. This unique name is represented by an integer according to the character set containing the character.

Tag: **CHAR$**

Attributes

CHAR.Dialect: Atom -- *Encoded with the ISO3166 two-letter-plus-digit identification scheme*

CHAR.Characterset: Atom -- *e.g. ISO646, XeroxCharacterset*

Content: SequenceOf Integer -- *Identifies characters in the Characterset*

The Dialect attribute identifies the language the character belongs to. An editor might use this information to select an appropriate hyphenation algorithm or to determine the correct sort sequence for an index (e.g. words beginning with ch collate between c and d in Spanish).

The Characterset attribute identifies the character set containing the character. The character set determines which characters are available and how those characters are encoded as integers. Interscript does not require certain character sets; any character set may be chosen. Of special importance is the Xerox Character Set [Reference 6]. It is a superset of other international standards and even includes the Japanese national standard JIS C6627.

To make the publication encoding easier to read, the character-integers are presented in legible form. Instead of integers, the corresponding characters appear enclosed in angle brackets.

Example 7.3: Equivalent character nodes using angle brackets.

```
--1--   1) {CHAR$ CHAR.Characterset = XeroxCharacterCode 63₁₆ }     is equivalent to
--2--      {CHAR$ CHAR.Characterset = XeroxCharacterCode <c>}
--3--   2) {CHAR$ CHAR.Characterset = ISO646 63₁₆ 61₁₆ 74₁₆}        is equivalent to
--4--      {CHAR$ CHAR.Characterset = ISO646 <cat>}
```

CHAR nodes serve two different purposes in a script. First, they act as *text-string values* for bindings, some of which have been described: user-sensible names, author, etc. In this case, they may contain several characters. Second, they define the *character content* of the document. In this case, they should contain only one character (see section 7.3.2). These two uses of the CHAR node are quite different; the distinction is important.

A CHAR node that is part of the character content of a document is a "leaf node" in the logical structure tree. As such, it can participate in layout. From the layout point of view, a CHAR node is a box (the character box) containing information about a certain distribution of ink within the box area. These character boxes are handled like all other boxes; laying out characters in a line is analogous to laying out lines in a column. It is their measures not their content that are relevant for layout.

How does a character become a box for the layout process? By augmenting a CHAR node with a FONT$ tag. Conceptually, a font can be considered to be a prototypical array of boxes available to an editor in a font database; a character-number can index into the array and obtain a box.

Tag: **FONT$**

Attributes

```
FONT.Name: Atom          -- Printer's name of the font family
FONT.Points: Number                          -- Body size of type
FONT.BaselineOffset: Number                  -- Baseline up-shift
FONT.Italic: Boolean     -- Use italic version of font if TRUE
FONT.Boldness: ONEOF ( lighter regular darker )
FONT.Strikeout: Boolean  -- Use 'strikeout' version of font
FONT.Underlining: ONEOF ( none single double )
                         -- The metrics for underlining are provided
                                                  by the font designer
```

The binding of the Name attribute describes the font. Hierarchical names containing a manufacturer's name may be used to designate fonts uniquely. Information about the font form (condensed, normal, or wide) or about the foreground and background color (normal or inverse) may also be included in the font name. Interscript does not standardize font naming: that is a matter for a separate standard. The discussion of font naming in the Interpress standard (XSIS 048201 ppg. 48-49) and the *Introduction to Interpress* (XSIG 038306 ppg. 70-93) are relevant.

The Size attribute defines the size of the font and the BaselineOffset attribute a vertical shift for superscripts and subscripts. Both attributes are expressed in points (one inch is approximately 72.27 points). Note that the attribute values are numbers: fractions are allowed and appropriate.

The Italic attribute, if TRUE, indicates that the italic version of the font should be used.

The boldness of a font is often included in the font name (e.g. Futura Light). The Boldness attribute is separate from this and may indicate "one step lighter" or "one step darker."

The Strikeout and Underlining attributes indicate whether the struck-out or underlined versions of the font should be used. These versions may actually exist in a font catalog or be synthesized: this is not of concern to Interscript.

Example 7.4: Character node for an **a**.

```
--1--   {CHAR$ FONT$ CHAR.Characterset = XeroxCharacterCode
--2--      CHAR.Dialect = US1
--3--        FONT.Name = Modern
--4--        FONT.Size = 14
--5--        FONT.Italic = TRUE
--6--        FONT.Boldness = darker
--7--        61₁₆ }
```

7.3.2 The CHARS$ encoding-notation

Since character nodes are treated individually during a pour, it is appropriate for the running text of a document to consist of *individual* CHAR nodes.

This is cumbersome in the publication encoding of Interscript. To make things easier to read, we define an encoding-notation.

A construction of the form {... CHARS$ ... <anything ...> ... } that lacks a MOLD$ tag—that is a node which syntactically looks like a CHARS node—is an encoding-notation for a sequence of CHAR nodes, each with a single character from the CHARS$ encoding. These CHAR nodes have whatever other tags and bindings the CHARS "node" has.

Example 7.5: Equivalent character strings using the CHARS notation.

```
--1--   {CHARS$ LABEL$                                          This CHARS
--2--   CHAR.Characterset = ISO646 LABEL.Labels = {A}           node . . .
--3--           <cat>}
--4--   CHAR.Characterset = ISO646 LABEL.Labels = {A}           . . . is equivalent to
--5--           {CHAR$ LABEL$ <c>}                              this sequence of
--6--           {CHAR$ LABEL$ <a>}                              CHAR nodes.
--7--           {CHAR$ LABEL$ <t>}
```

A construction of the form {... CHARS$ ... MOLD$ <anything ...> ...} —that is a "node" that syntactically looks like a "CHARS" mold—is an encoding-notation for a template which defines a repetition of CHAR nodes, with FILL nodes, PSEUDOCHAR nodes, and TAB nodes allowed.

Thus the encoding-notation {CHARS$ MOLD$} is an encoding for

```
{TEMPLATE$ TEMPLATE.Expresses = repetition
    {TEMPLATE$ TEMPLATE.Expresses = alternation
        {CHAR$ MOLD$} { FILL$ MOLD$} {TAB$ MOLD$} {PSEUDOCHAR$ MOLD$ } } }.
```

Note well that this CHARS encoding-notation does not augment Interscript in any way. It simply makes it easier and more compact to create the publication encoding.

7.3.3 Running text

Character content portions attached to the logical structure are handled as running text. The main problem with running text is to break it into lines. Interscript has no special tags to control line breaks. Whether a line should be broken on a word boundary or whether automatic hyphenation should be invoked is a local decision that a script does not control. The general penalty method (see section 7.5.3) could be applied to influence decisions as to how many character boxes are poured into a line. However, since penalties occuring at low document levels greatly complicate the formatting process, they are not normally used in lines. Each Interscript system is expected to know how to handle line formatting.

The pour operation concatenates all the character content portions of a stream (see section 7.5.1) and lets a line breaking algorithm assemble lines. The line assembly process cuts and rearranges the content portions as shown in Example 7.6:

Example 7.6a: Content portions in running text.

```
--1--   {CHARS$ <The quick bro>}
--2--   {CHARS$ <wn fox jumped.>}          This running text might be laid out as ...
```

Example 7.6b: Content portions laid out in lines.

```
--1--   {LINE$ ... {CHARS$ <The>}}                    Blanks at line
--2--   {LINE$ {CHARS$ <quick brown fox>}}             ends are omitted
--3--   {LINE$ {CHARS$ <jumped.>} ...}
```

This is the principle. In practice there are some irregularities connected with a line break: some content portions receive no box so they are not rendered, other content portions are inserted as box content without a corresponding portion in the running text.

- A discretionary hyphen may appear in running text. A discretionary hyphen defines a potential hyphenation point that may be used when a line break would occur "nearby." If the discretionary hyphen does not appear at the end of a line, no character box is attached to it and it is not rendered. Discretionary hyphens may be inserted by a user to correct automatic hyphenation errors or as a kind of manual hyphenation. They could also be inserted by an editor to remember hyphenation points produced by an automatic process.
- A discretionary hyphen may be inserted in the layout. In this case the hyphen character is only attached to a leaf box in the layout structure. This hyphen may be discarded when the text is reformatted.
- Hyphenation can even cause changes in the appearance of some characters in some languages. There are two such instances in German. In such cases, some content portions are added and only attached to the layout structure. They may be discarded when the text is reformatted.
- When words must not be separated by line breaks, a non-breaking space character can be used.
- When a space character is to be placed in a line that is already full, it is not displayed. The space character remains in the running text but is not attached to a box of the layout structure.

7.3.4 Tabs

A TAB node tells the formatting process that the actual content portion must be positioned at a tab stop (some character sets provide a special character for tab; this character may be used instead of a TAB node).

Tag: **TAB$**

The properties of that stop (e.g. the position or the alignment) are not defined as TAB attributes but as TABSTOP attributes of a particular TABSTOP node (see sections 7.4.6 and 7.4.7). TABSTOP nodes are defined as a sequence bound to the LINE.Tabstop attribute of the actual LINE node. The correspondence between tabs and TABSTOP nodes is given by a "counting off" process. Sequential TAB$ tags are counted; the n^{th} TAB$ is associated with the n^{th} TABSTOP node.

If text has already been placed at or beyond the tab position, overwriting does not occur. The formatting process starts a new line and the content portion is aligned at the appropriate position. This is particularly advantageous when a font larger than the one specified by a script's creator is being used because of a font substitution.

Typewriter-like tabbing is possible if an editor transforms typewriter tabs into Interscript tabs (and vice versa).

7.3.5 Textfields

Suppose a node contains an integer. How should that integer be rendered: with a decimal point, commas between triads, with a plus or minus sign? Since it is not a CHAR node, it cannot be rendered until the integer is transformed into one or more characters. And that character string must maintain its relationship with the original integer so changes to the integer produce a new character string. (The character string itself may not be changed since it has no identity of its own; it's simply the rendering of an integer.)

The TEXTSHOW$ tag is used with the CHAR$ tag to indicate that the characters in the content do not express the full meaning of the node; they are a representation of some other value. Consequently the textual content of a node with a TEXTSHOW$ tag should not be changed directly. The underlying value may be changed, of course, and that will cause a change in the character string. An editor that does not "understand" the TEXTSHOW$ tag, but does "understand" the CHAR$ tag will be able to display the textual representation, but not change it. This is an important feature of a TEXTSHOW node.

Tag: **TEXTSHOW$**

Attributes

TEXTSHOW.Type: OneOf (text amount date)
TEXTSHOW.Value: Any
TEXTSHOW.Picture: CHAR node -- *A COBOL picture clause if Type = amount*

Required tags: CHAR$

The Value attribute of the TEXTSHOW$ tag is the underlying value, e.g. an integer. The TEXTSHOW node may be augmented with a REMOTEBINDING$ tag; if it is, the value is effectively copied from another node.

The Type attribute requires the value of the textfield to be of a certain "type" (in this version text, amount, and date). This is of special importance when the value is copied by a REMOTEBINDING$ tag.

The Picture attribute determines how the value is to be transformed into a character string. This transformation depends on the tag of the Value and is automatically executed by an editor. Interscript version 1 does not define a transformation of its own. Instead, it uses the picture clause of COBOL that supplies a transformation of a number into text. Future Interscript versions may provide more possibilities for rendering numbers.

Example 7.7a: A textfield definition.

```
--1--   {CHARS$ TEXTSHOW$
--2--       TEXTSHOW.Type = amount TEXTSHOW.Value = 41
--3--       TEXTSHOW.Picture = {CHAR$ <9.9> }}          This textfield gets
                                                        evaluated to ...
```

Example 7.7b: An evaluated textfield.

```
--1--   {CHARS$ TEXTSHOW$
--2--       TEXTSHOW.Type = amount TEXTSHOW.Value = 41
--3--       TEXTSHOW.Picture = {CHAR$ <9.9> }
--4--       <4.1>}
```

7.3.6 "Pseudo" characters

The PSEUDOCHAR$ tag may be added to content portions. The formatting process will handle a PSEUDOCHAR node as though it were a CHAR node. If a box in a layout definition must be filled by the content of CHAR nodes, PSEUDOCHAR nodes may also be used.

The PSEUDOCHAR$ tag does not add any Interscript functionality to a node. It expresses a user's intent to treat a content portion as a character, and helps an editor perform operations like "select the next character." A PSEUDOCHAR$ tag might be attached, for example, to a logotype contained in a small bitmap frame.

Tag: **PSEUDOCHAR$**

Required tags: BOX$

7.3.7 Other content architectures

A fragment of Interpress encoding may be contained in a document as a content portion as identified by the INTERPRESS$ tag.

Tag: **INTERPRESS$**

Attributes

INTERPRESS.Version: Atom

Content: SequenceOf Integer -- *Probably an encoded sequence*

7.4 Layout constructs

This section discusses all the constructs that are needed to describe the geometrical layout. These constructs are used for the layout definitions in the processable format as well as the layout structure in the image format.

7.4.1 Measures

Measures define the size and positioning of boxes. They are expressed in terms of a Basic Measurement Unit (BMU). Within Interscript a BMU is a mica (0.01 millimeter).

Each measure is bound either to the atom synthesized, to the atom match, or to a node with the tag MEASURE$. The latter is called a numeric measure.

Tag: **MEASURE$**

Attributes

MEASURE.Under: Number
MEASURE.Nominal: Number
MEASURE.Over: Number

A MEASURE node is a record defining a variable measure. It supplies a Nominal value for a distance, together with an Over and an Under value. The Nominal value is a suggestion that the layout process should use when possible. It may be adjusted with some leeway, given by the Under and Over value. The Under value is always less than the Nominal value and defines a *lower limit* that must always be respected; its difference compared to the Nominal value gives the *shrinkability* of the measure. The Over value is always greater than the Nominal value and may be exceeded; the relevant information is its difference to the Nominal value that gives the *stretchability* of the measure.

Such variable measures are like *glue* in Knuth's TEX [reference 1]. They may be considered as springs that the layout process stretches or squeezes in order to get boxes into a containing box. For example, the layout process may squeeze character boxes in a line to avoid word wrapping.

When a measure must be fixed, all three values must coincide. All measures that occur in the layout structure† of the image format are fixed measures. When a measure is to take arbitrary values, the Under and Over values must specify the lowest and the highest values respectively. In this document, the following abbreviations are used:

- for *variable measures*: {MEASURE$ n1 n2 n3} instead of
  {MEASURE$ MEASURE.Under = n1 MEASURE.Nominal = n2 MEASURE.Over = n3}
- for *fixed measures:* {MEASURE$ n} instead of
  {MEASURE$ MEASURE.Under = n MEASURE.Nominal = n MEASURE.Over = n}

In the following example, the highest value is assumed to be 99999.

Example 7.8: Some measures.

```
--1--   {MEASURE$ -100 500 1000}          A variable measure
--2--   {MEASURE$ 1000}                   A fixed measure
--3--   {MEASURE$ 1000 1000 1000}         An equivalent fixed measure
--4--   {MEASURE$ 0 10 99999}             An arbitrarily stretchable measure
--5--   {MEASURE$ -99999 0 10}            An arbitrarily shrinkable measure
--6--   {MEASURE$ 0 0 100}                A non-shrinkable measure
--7--   {MEASURE$ -100 500 500}           A non-stretchable measure
```

The atom synthesized may be used only for the size measures. Synthesized size measures let a box assume a size dictated by the layout requirements of its content. When all subboxes are determined, the layout process replaces the synthesized size measures by numeric ones (see Chapter 8).

† The layout structure is a sequence of box-trees each of which has a page box as a root node. Chapter 8 contains a comprehensive discussion of the formatting process.

The atom match may be used for measures (or any other attribute) of nodes containing a MOLD$ tag. Match arrangements are discussed in conjunction with pouring in section 7.5.1.

7.4.2 Boxes: external view

All geometrical layout is defined in terms of boxes within a document presentation medium. A box is a rectangular area with its own cartesian coordinate system. The coordinate axes are named x and y. The positive x-axis, when rotated 90° counterclockwise, produces the positive y-axis. Both axes run parallel to the edges of the box.

Tag: **BOX$**

Attributes

BOX.X: SPAN node
BOX.Y: SPAN node
BOX.Rotation: Number -- *Must be 0, 90, 180, 270 ccw*
BOX.Clips: Boolean

Tag: **SPAN$**

Attributes

SPAN.LowPartExtent: MEASURE node | synthesized
SPAN.HighPartExtent: MEASURE node | synthesized
SPAN.FromLowContainer: MEASURE node
SPAN.FromHighContainer: MEASURE node
SPAN.FromLowSibling: MEASURE node
SPAN.FromHighSibling: MEASURE node

The size of a box is defined by the distances of the coordinate origin to the edges. More precisely:

the LowPartExtent measures give the distances to the lower edges and

the HighPartExtent measures give the distances to the higher edges.

Each measure has to be defined separately for the x- and y-direction (to simplify the description, the expressions "lower" and "higher" are also used here in the sense of "left" and "right" respectively).

Boxes can be **nested** to build a complex layout. Character boxes may fill line boxes, line boxes may fill paragraph boxes, and paragraph boxes may fill page boxes. The edges of boxes must be parallel to the edges of a containing box. Each box carries information about how it should be placed inside a containing box. It prescribes the distances

of its upper edge to the lower edge of the upper sibling box by the FromHighSibling measure,

of its lower edge to the upper edge of the lower sibling box by the FromLowSibling measure,

of its upper edge to the upper edge of the containing box by the FromHighContainer,

of its lower edge to the lower edge of the containing box by the FromLowContainer.

Each measure is separately defined for the x- and y-direction. So there are six measures for each coordinate direction associated with a box. These box measures are evaluated by built-in layout methods (see section 7.4.4). The particular layout method determines which measures are relevant. For each direction the box measures are collected in a SPAN node and are bound to the BOX attributes XSpan and YSpan. Figure 7.1 illustrates these measurements.

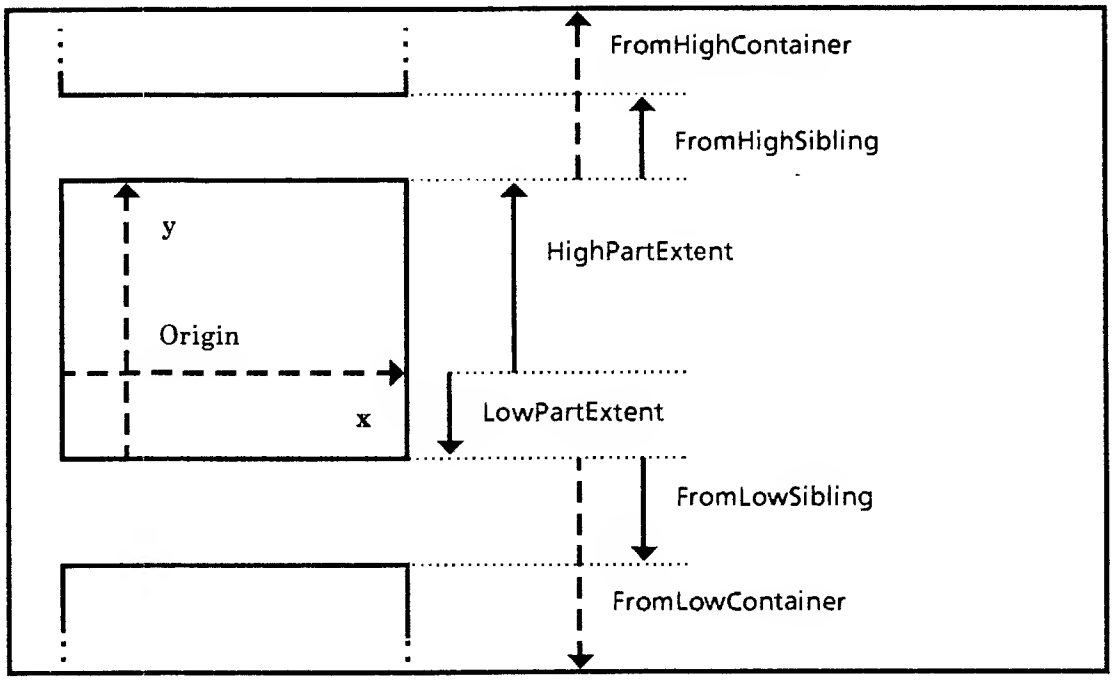

Figure 7.1 A box SPAN for the y-direction.
(Rotate ninety degrees clockwise to produce the x-direction figure)

Two further attributes influence the layout of a box. A box can be rotated when it is placed in a containing box. This rotation is measured by the angle between the positive x-axis of the containing box and the positive x-axis of the filled-in box. Since the edges of the filled-in box must be parallel to those of the containing box, only multiples of 90° are permitted. The BOX attribute Rotation defines the appropriate value. To keep explanations simple, coordinate rotations are ignored in this document.

Boxes may extend beyond the area of their containing box, perhaps by having a negative FromLowContainer or a large extent. The BOX attribute Clips controls rendering. If the Clips binding is FALSE, any portions that extend are rendered. If the Clips binding is TRUE, such portions are not rendered.

Boxes with variable measures, Rotation attribute, and Clips attribute are used in the layout definitions of the processable format. They provide rules about how to adapt positions and sizes of boxes to the filled-in content and the surrounding boxes.

Boxes in the layout structure of the image format have fixed positions and sizes. They are specified by their extent measures and the lower container distance for each coordinate direction. The other SPAN and BOX attributes are not relevant.

7.4.3 Color of boxes

The INKED$ tag determines the background color of boxes. Boxes without an INKED$ tag are transparent (i.e. an overlaying box does not cover the underlaying box). The ink provided by the INK$ tag is opaque and makes underlaying boxes invisible.

Note that overlaying does not depend on the box hierarchy. It is determined by the order in which the boxes are created by the formatting process; subsequent boxes overlay previously created ones.

Tag: **INKED$**

Attributes

INKED.Color: OneOf (black white)

Required Tags: BOX$

The value-space of the Color binding will probably be extended to include greys and colors.

7.4.4 Boxes: internal view

All but the leaf boxes in the layout structure have additional layout within them. They carry an INSIDELAYOUT$ tag whose attributes describe how the inside layout is to be performed.

Tag: **INSIDELAYOUT$**

Attributes

INSIDELAYOUT.X: INSIDELAYOUTMETHOD node
INSIDELAYOUT.Y: INSIDELAYOUTMETHOD node

RequiredTags: BOX$

The attributes are X and Y, which are bound to an INSIDELAYOUTMETHOD node. This node selects a method for calculating the position of a filled-in box (see the next chapter). Because boxes are independently laid out in the x- and y-direction, these nodes may be defined independently of each other.

Tag: **INSIDELAYOUTMETHOD$**

Attributes

INSIDELAYOUTMETHOD.Direction: OneOf (fixed up down onOrigins)
INSIDELAYOUTMETHOD.SiblingAdjacency: OneOf (parallel serial)

The attributes of the INSIDELAYOUTMETHOD$ tag are Direction and SiblingAdjacency. The Direction attribute can be bound to one of four predefined methods:

- **The fixed method**

  Each subbox to be poured defines its position by its two container distances; all other distances are ignored. For that, the extent measures of the containing box must already exist; containing boxes with synthesized extent measures are excluded from fixed inside layout. Normally only one container distance is specified and the other is given an arbitrary leeway. The formatting process will calculate the appropriate value. A typical example is the partition of a page into fixed boxes for a heading, a footing, and content. Here the fixed method is used for the x- and y-direction. The binding of SiblingAdjacency is ignored when the Direction is fixed.

- **The onOrigins method**

  This is also a "fixed" method. The origin of each box as it is placed into a containing box is made to coincide with the origin of the containing box. Sibling distances and container distances are ignored. Characters, for example, are normally filled into lines such that their base lines are at the same height (i.e. the onOrigins method is used for the y-direction). Of course, the layout method in the x-direction is not onOrigins as that would cause every character to be placed on top of every other. The binding of SiblingAdjacency is ignored when the Direction is onOrigins.

- **The up method**

  This is the "fluid" case. The first box to be poured defines its position by its FromHighContainer distance. Its other distance measures are ignored. The positions of subsequent boxes are calculated from the lower sibling distance of the preceding box and the higher sibling distance of the next box. The container distances are ignored.

  If those sibling distances are not identical, a compromise is necessary. The Sibling-Adjacency attribute of the INSIDELAYOUTMETHOD$ tag permits a choice between two compromise methods: serial which simply calculates the arithmetic average and parallel which takes into account the different stretchability of both distances. Typical examples are lines filled with character boxes according to the up method in the x-direction.

- **The down method**

  This is the up-method in the opposite direction. Typical examples are columns or pages filled with lines according to the down method in the y-direction.

The following two examples demonstrate the utility of variable measures in connection with inside layout methods. Example 7.9 shows how to center (in the x-direction) a box within a containing box.

Example 7.9: A centered box.

```
--1--   {BOX$ INSIDELAYOUT$
--2--       INSIDELAYOUT.X = {INSIDELAYOUTMETHOD$
--3--                           INSIDELAYOUTMETHOD.Direction = fixed} .....
--4--       {BOX$ BOX.X = {SPAN$ SPAN.FromLowContainer = {MEASURE$ 0 0 99999}
--5--                           SPAN.FromHighContainer = {MEASURE$ 0 0 99999}
--6--                           .....} }
```

Since the layout method of the outer box is fixed, only the container distances of the subbox are relevant for centering. Because the two container distances are of equal stretchiness, the formatting process will stretch them equally. This produces a centered box.

Example 7.10 handles the problem of breaks. This problem occurs when a box is filled according to the up or down method. When a pour operation encounters certain boxes ("breaking boxes"), it must stop filling the actual containing box and place such a breaking box into the next container (e.g. the first box of a chapter is always poured into a new page or column). Example 7.10 is a solution that utilizes the sibling distance (other solutions may utilize the FILL$ tag).

Example 7.10: A breaking box for the down method.

```
--1--   {BOX$ BOX.Y = {SPAN$
--2--                   SPAN.FromHighSibling = {MEASURE 99999}
--3--                   SPAN.FromHighContainer = {MEASURE 0 100 500} ... }
```

If there is a prior sibling box, this breaking box cannot be poured into a containing box according to the down method since it reqires an "infinite" higher sibling distance. But if this breaking box is the first box that is poured, the sibling distance is ignored; the breaking box will then be poured according to its higher container distance.

7.4.5 Page boxes

A page box is a BOX node carrying an additional PAGE$ tag. The attributes of the PAGE$ tag assemble the information about the medium on which a single document page is displayed. They are a subset of the page attributes of Interpress [Reference 2].

Tag: **PAGE$**

Attributes
PAGE.MediumName: Atom
PAGE.MediumXSize: Number
PAGE.MediumYSize: Number

RequiredTags: BOX$

The MediumName attribute identifies the desired medium. If it is bound to NIL, the document may be printed on a medium that is the default medium of the printer. As with Interpress, Interscript doesn't define other media identifiers. The MediumXSize and

MediumYSize attributes define the size of the medium in BMUs (micas, or 0.01 millimeters).

Page boxes are the starting point for the layout procedure that fixes box measures. Therefore the page box itself must have fixed extent measures (its distance measures are not relevant). The resulting layout structure is then a sequence of box-trees each of which has a page box as a root node.

7.4.6 Lines

A line is a node carrying the LINE$ tag. It is used in templates. The attributes of the LINE$ tag provide information about the appearance of a line.

The LINE$ tag does not extend the functionality of Interscript: it is the equivalent of a certain arrangement of boxes ("line boxes"). This equivalence is provided in the Interscript Standard as the formal definition of the LINE$ tag.

Although it does not add functionality to Interscript, there are two important reasons for the existance of the LINE$ tag:

- An editor can recognize lines during editing and therefore perform operations such as "select next line."
- An editor can recognize lines during formatting and therefore invoke special procedures for line layout and line breaking. Since these procedures are "hard-coded" into the editor, they speed up the layout process.

Tag: **LINE$**

Attributes

LINE.Justification: Boolean
LINE.Raggedness: ONEOF (atBeginning centered atEnd)
LINE.LeftMargin: MEASURE node
LINE.RightMargin: MEASURE node
LINE.InterlineLeading: MEASURE node
LINE.AboveBaseline: MEASURE node
LINE.BelowBaseline: MEASURE node
LINE.Tabstops: { SequenceOf TABSTOP nodes }

The Justification attribute determines whether the initial and final characters in a line are forced to be at the ends of the line box, with stretching occurring between character boxes.

The Raggedness attribute determines how unused space in a line should be distributed. It offers a selection between atBeginning (lines assembled from left to right would be "ragged" at the left), centered (lines would be "ragged" at both ends), and atEnd (lines assembled from left to right would be "ragged" at the right). Raggedness makes sense only for non-synthesized line widths. If Justification is bound to true, it overrules Raggedness.

The LeftMargin and the RightMargin attribute determine the line margins in BMUs. The left margin defines the FromLowContainer and the right margin the FromHighContainer distance.

The InterlineLeading attribute defines both sibling distance measures for the y-direction (they are equal).

The AboveBaseline and the BelowBaseline attribute define the line's HighPartExtent and the LowPartExtent measure in the y-direction.

The Tabstops attribute defines a sequence of TABSTOP nodes that apply to this line (see section 7.4.7).

7.4.7 Tabstops

The sequence of TABSTOP nodes bound to the Tabstops attribute of the actual LINE node contains information used by the formatting process when one or more consecutive TAB$ tags are encountered in a script (see section 7.3.4).

---

tag: **TABSTOP$**

Attributes

TABSTOP.Type: OneOf (left centered right aligned)
TABSTOP.AlignedOn: CHAR node
TABSTOP.Position: Number

The Position attribute specifies the distance from the beginning of the line to the tabstop expressed in BMUs. Note: This distance is NOT measured from the edge of the page.

The Type attribute determines whether the text following a TAB$ sequence must:

- begin at the tab position (Type is bound to left);
- end at the tab position (Type is bound to right);
- be centered around the tab position (Type is bound to centered);
- be moved such that a certain character (defined by the AlignedOn attribute) falls at the tab position (Type is bound to aligned). Aligned tabs may be used to define "decimal tabs."

Example 7.11a: Lines with TABSTOP nodes.

```
--1--   LINE.Tabstops =
--2--       {   {TABSTOP$ TABSTOP.Type = left
--3--               TABSTOP.Position = 30*254}
--4--           {TABSTOP$ TABSTOP.Type = aligned
--5--               TABSTOP.Position = 50*254
--6--               TABSTOP.AlignedOn = {CHAR$ <:>}  }
--7--   {LINE$ .....
--8--       {TAB$} {CHARS$ <Arrival>}
--9--       {TAB$} {CHARS$ <1:00 p.m.>}}
--10--  {LINE$ .....
--11--      {TAB$} {CHARS$ <Departure>}
--12--      {TAB$} {CHARS$ <10:00 p.m.>}}
```

Example 7.11b: Rendered lines with tabs.

```
--1--                   Arrival          1:00 p.m.
--2--                   Departure        10:00 p.m.
```

7.4.8 Fill

It is often necessary to terminate the "filling" of the layout entity currently being filled and then continue with the "next" one (e.g. when forcing a new line or new page). The FILL node aids in the implementation of this function (some character sets provide special control characters for creating a new line or page; those control characters may be used instead of FILL nodes).

Tag: **FILL$**

Attributes

FILL.Container: Atom | Nil

The FILL$ tag simply augments the node where it appears. It provides the relevant binding FILL.Container. The molds of a pour operation must be arranged to provide the functionality of terminating the filling of a layout entity. If FILL.Container is Nil, the entity whose filling is to be terminated is the immediate container; if FILL.Container is non-Nil, the entity is the first entity higher in the layout with a label matching FILL.Container.

7.4.9 Balanced nodes

A popular layout arrangement is balanced columns. The desired effect is to divide document content evenly between two or more layout boxes, where "evenly" is in terms of set-length.

The appropriate way to accomplish this within the Interscript layout facilities is to enclose the "balanced" layout boxes within an outer box, with their fromContainer measures arranged so they must all be the same height as the containing box.

Two column boxes might be contained within a page-area box. The column boxes are of variable height (their extent measures have a lot of stretch). The page-area box is also

stretchy. But the column boxes' fromLowContainer and fromHighContainer measures are zero. Thus the two columns are forced to the same height.

But such an arrangement can be difficult for an editor to paginate. The user-intent is not clear and an editor might try, and then reject, many different lines-per-column arrangements before it finds an appropriate one.

The BALANCED node is provided as an aid to an editor.

Tag: **BALANCED$**

If two or more sibling nodes have a BALANCED$ tag, then: 1) they all have an INSIDELAYOUT$ tag and the same InsideLayout attributes; 2) their InsideLayout method, for exactly one of X and Y, has Direction = up or Direction = down; 3) they are constrained to have the same total extent in the X or Y direction.

7.4.10 Templates

The rules that define layout in Interscript allow a page to vary with the formatted content. The construction rules for the layout structure are given by the POUR.Template binding in a pour node. Within that binding, it is necessary to allow flexibility: to allow "some number of these" or "this or that, whichever matches the logical content." The flexibility is provided by TEMPLATE nodes within the POUR.Template.

Tag: **TEMPLATE$**

Attributes

TEMPLATE.Expresses: OneOf (sequence alternation repetition)

Content: SequenceOf Node

The Expresses attribute gives the semantics of the node:

- sequence
  All the subnodes of the TEMPLATE node must be chosen for the layout and must appear in the order given.
- alternation
  One of the subnodes of the TEMPLATE node must be chosen.
- repetition
  The TEMPLATE node contains only one subnode that may be repeated arbitrarily often.

Example 7.12 provides a template for a report layout. It consists of a special first page and then some number of following pages. Each page contains a heading, a content box, and a footing.

Example 7.12 The template of a report (incomplete).

```
--1--   {TEMPLATE$ TEMPLATE.Expresses = sequence
--2--      {BOX$ PAGE$ TEMPLATE$ TEMPLATE.Expresses = sequence      First page
--3--            {BOX$ MOLD$ ..... }                                 heading
--4--            {BOX$ MOLD$ ..... }                                 content box
--5--            {BOX$ MOLD$ ..... }}                                footing
--6--      {TEMPLATE$ TEMPLATE.Expresses = repetition
--7--         {BOX$ PAGE$ TEMPLATE$ TEMPLATE.Expresses = sequence   Following pages
--8--            {BOX$ MOLD$ ..... }                                 heading
--9--            {BOX$ MOLD$ ..... }                                 content box
--10--           {BOX$ MOLD$ ..... }}}}                              footing
```

Example 7.13 shows a page template that allows a choice between a two-column and a one-column page.

Example 7.13 A page template (incomplete).

```
--1--   {TEMPLATE$ TEMPLATE.Expresses = alternation
--2--      {BOX$ PAGE$ TEMPLATE$ TEMPLATE.Expresses = sequence     Two-column page
--3--         {BOX$ TEMPLATE$ TEMPLATE.Expresses = repetition        left column
--4--            {LINE$ .... }}
--5--         {BOX$ TEMPLATE$ TEMPLATE.Expresses = repetition        right column
--6--            {LINE$ .... }}}
--7--      {BOX$ PAGE$ TEMPLATE$ TEMPLATE.Expresses = repetition    One-column page
--8--         {LINE$ .... }}}
```

7.5 Pouring constructs

In the processable format, the logical structure with the content and the layout definitions are separated. Therefore the formatting process must include functions that recombine them so the text is positioned in the right place on a page. These functions are the pour operations. This section describes the tags controlling these pour operations.

7.5.1 POUR and MOLD nodes

The formatting process performs the layout of a document by choosing boxes from a template, filling them with text, and arranging them within containing boxes. It is a nested bottom-up process consisting of a hierarchy of independent pour operations each governed by a single POUR node. Its result is a sequence of boxes which, in turn, may be used as input for a higher pour operation.

A POUR node may contain content portions and subordinate POUR nodes providing boxes that are constructed by the corresponding pour operations. The content portions and these boxes are called content nodes. A pour operation pours the content nodes into some containing boxes.

Tag: **POUR$**

Attributes

POUR.Labelset: { SetOf Atom}
POUR.Template: Node
POUR.SatisfactionThreshold: Number
POUR.SatisfactionForwardSearch: Number
POUR.SatisfactionUpwardSearch: Number

The SatisfactionThreshold, SatisfactionForwardSearch, and SatisfactionUpwardSearch attributes are discussed in connection with the PENALTY$ tag (see section 7.5.3).

The Labelset attribute defines the set of *relevant labels*. They are used to indicate in which boxes the content nodes must be poured. These boxes are called MOLD nodes. The correspondence between MOLD nodes and content nodes is defined in two steps (the next chapter gives a more detailed description):

1. Each subset of relevant labels defines a stream of MOLD nodes and a stream of content nodes taking in all nodes that are labelled with the corresponding label set and whose other labels are not relevant. The pour operation concatenates the members of a stream in the order in which they appear in the POUR node to produce a solid layout. Since the pour operation handles each stream independently, the succession of streams is not relevant and the members can be mixed.
2. If a stream of liquid nodes and a stream of MOLD nodes have the same subset of relevant labels, the n^{th} liquid node is poured into the n^{th} MOLD node.

Example 7.14 demonstrates how the set of relevant labels defines streams. This example uses only content portions.

Example 7.14 Streams in a POUR node.

```
--1--   {POUR$ POUR.Labelset = {English French Footnote}
--2--      {  {CHARS$ LABEL$ LABEL.Labels = {English} <English text 1>}
--3--         {CHARS$ LABEL$ LABEL.Labels = {English Footnote} <English footnote>}}
--4--      {CHARS$ LABEL$ LABEL.Labels = {French} <French text 1>}
--5--      {CHARS$ LABEL$ LABEL.Labels = {English} <English text 2>}
--6--      {  {CHARS$ LABEL$ LABEL.Labels = {French} <French text 2>}
--7--         {CHARS$ LABEL$ LABEL.Labels = {French Footnote} <French footnote>}}}
```

The above POUR node contains four streams given by the subsets of {English French Footnote}:

- {English} defines the stream {CHARS$ <English text 1>}{CHARS$ <English text 2>}.
- {French} defines the stream {CHARS$ <French text 1>}{CHARS$ <French text 2>}.
- {English Footnote} defines the stream {CHARS$ <English footnote>}.
- {French Footnote} defines the stream {CHARS$ <French footnote>}.

POUR nodes contain templates that define construction rules for the layout structure. The templates are bound to the Template attribute. Often they are predefined in the global environment or in styles and attached to certain logical "types." For example, a chapter-

template may be predefined and copied into those POUR nodes that should be handled as "chapters." (Note: at present a CHAPTER$ tag as a logical structure cannot be defined.)

Tag: **MOLD$**

Attributes

MOLD.Coercion: QuotedExpression

The Coercion attribute enables changing attribute bindings after a content node has been poured into the MOLD node. A typical use of Coercion is to handle a pour specifying that the first letter of a line is to be two points larger than the other letters. This larger font cannot be stated in the logical content since it is not known which characters will be placed at the beginning of a line. Also, this larger font cannot be stated in the layout definition of the line since the font of the poured-in characters is not known. Example 7.15 shows a solution to this problem utilizing Coercion. Chapter 8 discusses Coercion more thoroughly.

Example 7.15 A MOLD node with Coercion.

```
--1--   {LINE$ ...
--2--      {CHAR$ MOLD$ ...                          First character box
--3--         MOLD.Coercion = "FONT.Size = FONT.Size# + 2"}
--4--      {CHARS$ MOLD$ ... }}
```

7.5.2 VACUUM nodes

Sometimes the desired effect of a "pour" is not simply to put each item of content into a place in a layout. Often one wishes to replicate certain values from the logical structure. For example, a page heading given as content portion of the logical structure will appear in each page node as the result of a pour, yet there is only one instance of the page heading in the document.

This need is addressed via the VACUUM node.

Tag: **VACUUM$**

Attributes

VACUUM.Sources: {SetOf Atom }

Required Tags: MOLD$

When a VACUUM node is encountered in a template, the pour operation is at a certain node, matching the pour label set, in the content. The content tree is scanned upward, examining the direct ancestors of that node and their immediate descendants, for the first instance of a VACUUMSOURCE node whose binding to Sources is the same as in the VACUUM node. That is, a match with $(parent^i)_j$ for the smallest i, then the smallest j. The content of this node is used as the source of the pour into the VACUUM

node in the template. The pour may alter bindings and content in the target VACUUM node. If no matching VACUUMSOURCE node is found, the VACUUM node is treated just like an ordinary mold that lacks matching liquid.

Tag: **VACUUMSOURCE$**

Attributes

VACUUMSOURCE.Sources: {SetOf Atom }

7.5.3 Penalties

Templates often allow an editor to choose among several, valid choices. If there were no additional information, an Interscript editor would be free to choose any alternative which managed to lace molds and content in a one-to-one correspondence. Consider a simple example of lines that are to be filled into pages. When does the pouring process stop filling one page and begin another? If there is no specification, putting only one line on each page would be perfectly legal.

Such unrestricted choices are often given when formatting lines since it is assumed that each editor knows how to handle them. To disallow this freedom and to control the various choices, Interscript provides the PENALTY$ tag. The PENALTY$ tag is used in templates and may be attached to any node. Since its purpose is to discriminate among alternatives, it is normally attached to nodes that appear in a choice or a repetition.

Tag: **PENALTY$**

Attributes

PENALTY.Amount: Number

The Amount attribute is bound to a number defining a penalty. It is the task of the pour operation to choose alternatives such that the sum of all penalties is as small as possible. The alternatives of all hierarchy levels contribute to this penalty sum. Therefore a choice on a lower level must be *propagated to the whole document.* "Good" choices may prove to be "bad" when considered on a higher level.

When formatting the simple document in Example 7.16, an editor should create as few lines and pages as possible. Therefore each line and page gets a penalty.

Example 7.16: Penalties for a simple page layout.

```
--1--   {DOCUMENT$ POUR$ POUR.Labelset = {A}
--2--   POUR.Template = {TEMPLATE$ TEMPLATE.Expresses = repetition
--3--           {BOX$ PAGE$ INSIDELAYOUT$
--4--           PENALTY$ PENALTY.Amount = 2000
--5--               {TEMPLATE$ TEMPLATE.Expresses = repetition
--6--                   {LINE$ PENALTY$ PENALTY.Amount = 100 ...}}}}
--7--       {CHARS$ LABEL$ LABEL.Labels = {A} ... <some text>}}
```

To calculate the penalties in Example 7.16, we assume the total number of text lines to be 12. The optimal selection with respect to the inner template (line 5) is choosing only one line for a page which supplies a penalty sum 25200 for the whole document (2000 × 12 pages plus 100 × 12 lines). If the worst selection of the inner template is chosen (all lines on one page), the penalty sum is 3200 (2000 × 1 page plus 100 × 12 lines).

Contradictory conditions as in the above example often occur between different templates. It would be impossible to enumerate each of them and instruct an editor on its alternatives. The PENALTY$ tag provides a universal method to make all alternatives comparable. So a script doesn't explicitly define how it is to be rendered. The local editor must find a layout that minimizes the penalty. Different systems may produce different results because of their different minimizing algorithms. Since a local change of a penalty may have global effects, one must handle the penalty number carefully.

Minimizing the penalty requires examining an entire set of alternatives before a single alternative is chosen. Normally this set is large and it would take too much time to examine it completely. Therefore *approximation methods* are used as determined by the attributes SatisfactionThreshold, SatisfactionForwardSearch, and SatisfactionUpwardSearch of the POUR$ tag (see the next chapter).

A second tool is also useful to reduce the complexity of minimizing penalties. The FENCE$ tag permits a layout to be broken into a sequence of pieces with a "hard" boundary (or fence) between the pieces. The pouring process optimizes the penalty of one piece after another which is much faster than if it were handling the layout as a whole.

Tag: **FENCE$**

7.5.4 TOGETHER relations

A content portion must frequently be rendered near another. Typical examples are a caption, which must not be separated from the image with which it is associated, or a footnote, which should be rendered on the same page as its reference.

The requirements of a caption associated with an image can be "structurally" met by making the caption immediately follow the image in the same stream. One can simply define a containing box in which both the caption box and the image box are placed. Since a pour operation never divides boxes, those two boxes cannot be separated by a subsequent pour operation. This problem can thus be solved by the appropriate template definition.

A structural solution won't work for a footnote and its reference because they are in different streams. Since they are laid out independently, there must be additional information to relate them. That's the function of the TOGETHER$ tag.

Tag: **TOGETHER$**

Attributes

TOGETHER.Penalty: Number
TOGETHER.LevelLabel: Atom | Nil

Content: SequenceOf Node

All the subnodes contained in a TOGETHER node should be rendered as "near" as possible. The Penalty attribute defines a penalty for the "distance" between the TOGETHER subnodes. This distance counts how many boxes lie in the layout structure between the boxes containing the TOGETHER subnodes. All boxes are not necessarily counted. It doesn't matter, for example, if the boxes for a footnote and its reference lie in other boxes. Only page boxes count. Relevant boxes are distinguished by a label-atom bound to the LevelLabel attribute. The algorithm for determining such a distance is explained in the next chapter. This distance and the associated penalty will be calculated by the corresponding pour operation and added to the penalty sum of the layout.

7.5.5 Paragraphs

A PARA node collects together the text and layout information that make up a single document paragraph. The PARA$ tag does not extend the functionality of Interscript: a PARA node is the equivalent of a POUR node which "pours" characters (and other things) into lines. The PARA node has been standardized so editors can easily recognize the straightforward instances of paragraphs.

The PARA$ tag standardizes "reasonable" cases. Exceptional cases must explicitly use pour constructs and declare themselves as paragraphs by the PSEUDOPARA$ tag.

Tag: **PARA$**

Attributes

PARA.Justification: Boolean
PARA.Raggedness: ONEOF (atBeginning centered atEnd)
PARA.AvoidWidow: Boolean
PARA.AvoidOrphan: Boolean
PARA.FirstLineLeftMargin: MEASURE node
PARA.LeftMargin: MEASURE node
PARA.RightMargin:MEASURE node
PARA.Preleading: MEASURE node
PARA.Postleading: MEASURE node
PARA.AboveBaseline: MEASURE node
PARA.BelowBaseline: MEASURE node
PARA.InterlineLeading: MEASURE node
PARA.LabelAugment: Atom | Nil
PARA.Tabstops: { SequenceOf TABSTOP nodes }

The attributes specify the template that is associated with a PARA node. More precisely, they determine the corresponding attributes for the lines in that template. The following attributes have a special meaning.

The FirstLineLeftMargin attribute defines the left margin of the first line. The left margin of all other lines is defined by the LeftMargin attribute. The RightMargin attribute defines the right margin for all lines.

The InterLineLeading attribute defines the "from sibling" attributes for the lines. There are two exceptions: the FromHighSibling distance measure for the first line is given by the PreLeading attribute, the FromLowSibling distance measure for the last line by the PostLeading attribute.

The WidowControl and the OrphanControl attributes define whether widows (the first line of a paragraph that is separated from the following lines) and orphans (the last line of a paragraph that is separated from the preceding lines) should be avoided if possible.

Essentially, paragraphs are POUR nodes whose template provides "lines" into which paragraph content is poured. Two special labels are used as follows:

- The ParaContent label controls pouring the paragraph. It is the only pour label of this pour and is attached to the paragraph content and the MOLD nodes in the template lines.
- The Line label is attached to the resultant lines of the paragraph pour. It is expected that higher pour operations with Line as a pour label will handle those lines.

The Line label cannot distinguish between lines produced by different paragraphs. This is sometimes necessary, for example to cause the lines of a footnote paragraph to be directed to the footnote area of a page. Therefore the labelset of the resultant lines may be augmented by an additional label supplied by the LabelAugment attribute of the paragraph node. The result of the paragraph's pour is labelled with Line and the

LabelAugment value, if any. In principle, this result is a sequence repeating the following box:

{ BOX$ LABEL$ LABEL.Labels = {Line PARA.LabelAugment # } ....}.

For the precise definition of a paragraph in terms of a pour, please see the Interscript Standard.

7.5.6 "Pseudo" paragraphs

As with "pseudo" characters, the formatting process handles a PSEUDOPARA node as though it were a PARA node.

The PSEUDOPARA$ tag does not add any Interscript functionality to a node. It expresses a user's intent to handle a content portion as a paragraph and helps an editor perform operations as "expand selection to a paragraph." PSEUDOPARA$ tags will typically be attached to pour nodes, where the pour node is logically a paragraph, but utilizes some complex layout arrangement not expressable via an ordinary PARA node.

Tag: **PSEUDOPARA$**

POUR$ is not a required tag, but it will frequently be present. It is not required because a logical pararaph *could* be expressed without a pour: it could be a sequence of line nodes.

7.6 Styles

Many editors use "styles" in their user interface. Styles enable a user to associate a named property, like *foreign word*, with text. The style can then cause the text to be presented in some different way, such as in italic font. This indirection has two advantages: 1) different properties (*foreign word* and *emphasis*, say) may be rendered the same, yet can be discriminated in a search; 2) the presentation of a document may be altered by substituting a new set of styles.

The concept of styles exists within Interscript via indirection. The STYLE$ standard document construct defines the *user-sensible* styles of a document.

Tag: **STYLE$**

Attributes

STYLE.Expansion: Node | Quoted Expression

The binding to STYLE.Expansion is the expansion of the style. A style definition exists in a script by being bound to a name, such as Name = { STYLE$ ... }. Invoking this style within the script takes this form: Name#:STYLE.Expansion # | .

A user-sensible name is typically attached to a style by augmenting a STYLE node with a USERNAME$ or USERDESCRIPTION$ tag.

7.7 Non-Interscript editing

Interscript enables a non-Interscript conforming editor to "edit" a script by allowing it to leave "warnings" in nodes. This is particularly useful when a *user* wishes to edit a node that he understands but the workstation software does not; the workstation could allow him to directly edit Interscript constructs.

These warnings are given by CAUTION$ tags. These tags indicate that possibly non-Interscript conforming edits have been made to this node. Editors should exercise care. If an editor understands a node, it can, if it desires, remove the warning.

Tag: **CAUTION$**

7.8 Hints

It may be convenient for an editor to remember accelerator information in a script. An editor might, for example, remember the line-breaks associated with a paragraph and a particular set-width. This could speed a subsequent pagination operation.

Interscript provides this functionality via the HINT node. A HINT node carries redundant information which may be useful to an editor, presumably because the information is expensive to recompute.

Tag: **HINT$**

Attributes

HINT.Predicate: Boolean -- *Should be TRUE*
HINT.Identifier: Atom

Content: Any

A HINT node only contains information concerning its directly enclosing node and the nodes hierarchically contained within it.

If an editor alters a document, it must do something about HINT nodes that carry information about the edited fragment of the document that might no longer be valid. *It is always appropriate for an editor to delete a HINT node.* This is not considered an editing action.

The Identifier attribute uniquely identifies the nature of the hint. An editor that recognizes the identifier will be able to take advantage of the hint content and perhaps recreate it after an editing action. An editor that does not recognize the identifier will not be able to take advantage of the content; if it edits the document, it should delete the hint since it does not know how to recreate it.

The Predicate attribute appears for future compatibility. It should be set to TRUE.

7.9 Revisions

Documents are often revised and it is common to track the parts of a document that have been changed. Interscript enables this via the REVISION$ tag. The REVISION$ tag, applied to a node, indicates that the node and its content, or some fraction thereof, has been "revised." REVISION$ tags are optional. An editor may edit without utilizing REVISION$ tags; it can also place a REVISION$ tag on a node without setting revision information in all ancestor nodes.

Tag: **REVISION$**

Attributes

REVISION.Version: Integer

A typical editing action is when an editor removes all "old" revision indications, then attaches new revision indications. Another action is when an editor "ages" the older revision indications.

The Version binding allows for "aging" revisions. A "new" revision always gets a Version of zero. An editor may, as an editing action, "age" revisions by subtracting one from the Version of all pre-existing REVISION$ tags. Thus the revision versions within a document range from the newest revisions, with Version=0, to some old versions with negative Version bindings.

This REVISION$ tag capability is not the same as allowing a "last edit date" to be attached to an arbitrary node. A possible extension to Interscript is to allow multiple "edit date" values to be associated with the DOCUMENT node and have them correlated with revision Versions.

8

The formatting process

Chapter 7 lists the standard document constructs and formally defines them as tags with associated attributes. With those constructs we are now able to express documents as scripts. But if we received such a script, we still wouldn't know how to render it. We also need to know the meaning of the standard document constructs. This meaning is given by the built-in procedures invoked by those constructs. This chapter describes the built-in procedures controlling rendering; the Standard will provide their formal definitions. Note that these procedures are not described in the base language and are not interchangeable; they are expected to be "hard coded" in an Interscript editor.

8.1 Formatting process overview

The formatting process is part of an editor and is invoked each time a script is to be rendered (on screen or on paper). The formatting process reads internalized scripts and starts special actions when encountering layout or pour constructs (defined by the tags listed in section 7.5 and 7.6). It produces a new script that represents the image format of the original script.

The image format describes the layout of a script. It may contain nodes of any type, but only its BOX nodes will be rendered. Essentially it consists of a sequence of page boxes each of which contains a tree of nested subboxes. These box trees form the layout structure. The content portions of the original script are broken into parts and filled into the leaf boxes of the layout structure.

Interscript's description of the formatting process is declarative and not procedural. Interscript imposes rules which the resulting image format must satisfy, but does *not prescribe* the procedures by which legal image formats are achieved. In particular, Interscript does not standardize how to break text into lines or when to stop filling lines into a box. There is the general understanding that boxes should be filled as much as possible, although each Interscript editor is expected to apply its own filling algorithm.

Interscript provides its own formatting process model. This model does *not uniquely determine* the result and allows the construction of several legal image formats. It is used for checking Interscript editors: an image format produced by an Interscript editor is considered legal if it can also be produced by Interscript's layout process.

The task of the formatting model is to describe clearly and simply Interscript's formatting rules. That description should not be regarded as a model for implementation. To demon-

strate how an effective implementation might work, section 8.5 gives an example produced by an existing editor.

Interscript's formatting process proceeds in two major steps: pouring and fixing.

Pouring produces an intermediate script by copying the original script and replacing the POUR nodes by sequences of nested nodes that are constructed according to a POUR node's template. Normally pouring produces boxes in which the content of a document is filled. Those boxes are still stretchable and may change their size depending on their content and surroundings. They represent the node structure of the image format but lack a fixed arrangement. That intermediate script is called the incomplete image format. Pouring is described in detail in section 8.2.

Fixing occurs after pouring. It takes the incomplete image format and arranges the final box positions by calculating and fixing box measures. Its result is the complete image format. Fixing is described in detail in section 8.4.

An effective formatting process would not proceed this way. Rather than performing all pours and then detecting an error during fixing, it would perform a provisional fixing incrementally after each pour.

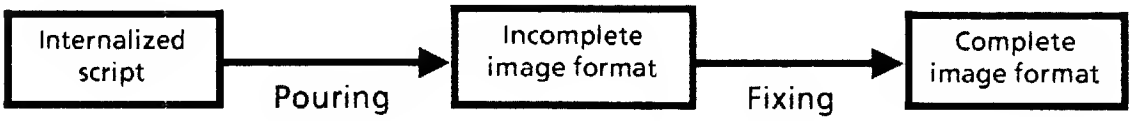

Figure 8.1: Interscript's formatting model

Normally templates of POUR nodes do *not uniquely* prescribe the nodes to be constructed as the result of the pour. They offer choices and repetitions which allow several image formats to be legal. For example, it may happen that rendering only one character on each page is a legal image format. To restrict this freedom and to express layout preferences of a script's originator (e.g. a text portion and an image should be rendered on the same page), penalties may be attached to layout decisions. In this case several pour processes must be started to get an image format with the least (or approximately least) penalty sum. Interscript's selection algorithm controls this process. It is described in detail in section 8.3.

8.2 Pouring

Interscript's pour process copies the original script and replaces all the POUR nodes. It is a nested process that runs through the dominant script structure in bottom-up fashion handling each POUR node separately (each such operation is called a *pour operation*).

Each pour operation proceeds in two steps:

1. A pour operation constructs a sequence of nodes that replaces the POUR node. The template associated with the POUR node provides the necessary construction rules. Normally a pour operation constructs BOX nodes.
2. A pour operation fills MOLD and VACUUM nodes with content. Such content may consist of elementary content boxes (such as character boxes) or of nodes produced by subordinate pour operations.

The result of a pour operation is a sequence of nodes with some internal structure. When all POUR subnodes of a POUR node have been poured, a pour operation for the POUR node

itself is started. The nodes produced by the subordinate POUR nodes are then part of the content of the higher POUR node. In this way the pour process continues upward in the dominant script hierarchy.

8.2.1 Creating nodes according to a template

A template is bound to the POUR.template attribute and consists of a tree of nodes. The pour operation traverses this tree in depth-first order. When *descending*, it creates image format nodes by transferring certain template nodes with all their tags and bindings into the image format preserving the hierarchical order. Each template node (except the leaf nodes) carries a TEMPLATE$ tag whose Expresses attribute tells the pour operation which of the subordinate template nodes it may transfer to the image format:

- Expresses bound to sequence.
  The pour operation must transfer each subnode.
- Expresses bound to choice.
  The pour operation must transfer one of the subnodes. The alternative to be chosen is not defined explicitly in a script. It is the task of an Interscript editor to find the "right" alternative (i.e. an alternative that allows successful formatting of the whole script). Normally the "right" alternative is not uniquely determined; several alternatives may prove to be "right." The pour operation at this point ignores penalties that may be attached to the alternatives (see section 8.3). The pour operation continues descending the template within the chosen alternative.
- Expresses bound to repetition.
  The pour operation can decide whether to repeat the inside node or not. A script doesn't explicitly limit the number of repetitions. It is the task of an Interscript editor to find the "right" number of repetitions. Again, the "right" number is normally not uniquely determined. The pour operation at this point ignores penalties that may be attached to the repetition node (see section 8.3). When the pour operation chooses a repetition, it continues descending the template within the repeated node. Otherwise it ascends to the parent node of the actual node.

Templates may contain content portions. They are associated with fixed boxes and will be rendered in these boxes (such as fixed heading text contained in a heading box). They do not belong to the logical structure.

Templates may contain POUR nodes. These POUR nodes do not belong to the logical structure but to the layout structure. They are handled like any other POUR node. Thus, a subordinate pour operation is started creating a sequence of nodes replacing the POUR node. POUR nodes in templates are used to put some content with dynamic layout rules into predefined layout boxes. For example, such POUR nodes allow defining paragraphs in headings or footings.

In this way the pour operation constructs a sequence of nested nodes that replaces the POUR node. In the second step, the pour operation traverses these nodes in depth-first order and fills VACUUM and MOLD nodes with content.

8.2.2 Filling MOLD nodes

MOLD nodes in the image format are placeholders for content nodes of POUR nodes. These content nodes may consist of elementary content boxes (such as character boxes) or of nodes produced by subordinate pour operations. The pour operation proceeds through these six steps:

1. The pour operation builds *streams of MOLD nodes* for each subset of the set of *relevant* labels (given by the LabelSet attribute in the actual POUR node). A MOLD

node belongs to such a stream if it is labelled with the corresponding label set and all its other labels are not relevant. The pour operation concatenates the members of a stream in depth-first order as they appear in the image format.

2. In the same way the pour operation builds *streams of content nodes*. It traverses the subnodes of the actual POUR node in depth-first order looking for leaf nodes (which are then elementary content boxes) or subordinate POUR nodes. In the first case it examines the set of labels of that leaf node looking for relevant labels. In the second case it examines the set of labels of the nodes resulting from the pour operation of that subordinate POUR node. The nodes found in this way are concatenated into streams.

3. The pour operation *compares the content node streams and the* MOLD *node streams*.

   - Streams associated with the same subset of relevant labels must have the same number of members. Partial pours cause pour errors.
   - Content node streams without a corresponding MOLD node stream and content nodes without relevant labels are appended to the result of the pour operation. Higher pour operations are expected to handle them. Otherwise they are discarded (e.g. footnotes may occur in the TOGETHER nodes with the referred text; but text and footnotes are handled by different pour operations).
   - MOLD node streams without a corresponding content node stream and MOLD nodes without relevant labels remain as they are.

4. If a content node stream matches a MOLD node stream, the n^{th} content box is associated to the n^{th} MOLD node. The pour operation *checks whether each content node fits the corresponding MOLD node*. For this, the content node must carry *all* the tags of the MOLD node (except the MOLD$ tag); otherwise a pour error occurs.

5. Fitting content nodes are transferred into the image format *replacing the corresponding MOLD nodes*. The relevant bindings of the MOLD node remain valid. *They overwrite the corresponding bindings in the content node* (except the binding of the MOLD.coercion attribute which is just added to the content node). Attributes in the MOLD node that are bound to the atom MATCH are excluded. Their bindings are determined by the content node. Note: since pour operations handle internalized scripts, the set of relevant bindings in a MOLD node is complete. If such a binding should get its value from a content node, it must explicitly be bound to the atom MATCH.

6. The pour operation *evaluates the MOLD.Coercion expression* within the transferred content node. This may cause an overwriting of bindings. It may even happen that the pour operation must start again from the beginning (e.g. perform new line breaks if the Coercion expression requires a larger font size making the text extend beyond the allocated line length).

8.2.3 Filling VACUUM nodes

The content of VACUUM nodes is not predefined in the template, but must be poured in from an appropriate VACUUMSOURCE node in the logical structure.

VACUUMSOURCE nodes and VACUUM nodes are associated by matching Source label-atoms. There may be several VACUUMSOURCE nodes associated with one VACUUM node. The pour operation must choose the "nearest" one. This is done by the following procedure:

When encountering a VACUUM node, the pour operation starts examining the children of the actual node which is to be filled in from left to right. If no appropriate

VACUUMSOURCE node is found among them, the pour operation ascends to the next higher node again examining the children. It continues this procedure until it finds an appropriate VACUUMSOURCE node or reaches the root node.

If no VACUUMSOURCE node is found, the VACUUMSOURCE node is treated like an ordinary MOLD box without a matching liquid node.

8.2.4 An example of a one-level pour

This section demonstrates the principles of the pouring process. It discusses the one-level POUR node shown in Example 8.1.

Example 8.1: One-level pour †

```
--1--   {POUR$  POUR.LabelSet = {Text}
--2--       POUR.Template =
--3--       {TEMPLATE$  TEMPLATE.Expresses = repetition                    Template
--4--           {BOX$  LABEL$  LABEL.Labels = {TextLine}                   Line box
--5--               {TEMPLATE$  TEMPLATE.Expresses = sequence
--6--                   {CHAR$  FONT$  MOLD$  LABEL$ ...                   First MOLD
--7--                       LABEL.Labels = {Text}                           character box
--8--                       FONT.Name = Roman  FONT.Points = 12             (with coercion)
--9--                       FONT.BaselineOffset = 0
--10--                      FONT.Boldness = MATCH
--11--                      FONT.Italic = MATCH  FONT.Strikeout = MATCH
--12--                      FONT.Underlining = MATCH
--13--                      MOLD.coercion = "FONT.Boldness = FONT.Boldness# + 1" }
--14--                  {CHARS$  FONT$  MOLD$  LABEL$ ...                   MOLD character boxes
--15--                      LABEL.Labels = {Text}                           (without coercion)
--16--                      FONT.Name = Roman  FONT.Points = 12
--17--                      FONT.BaselineOffset = 0
--18--                      FONT.Boldness = MATCH
--19--                      FONT.Italic = MATCH  FONT.Strikeout = MATCH
--20--                      FONT.Underlining = MATCH }}}}}
--21--       {CHARS$  FONT$  LABEL$ ...
--22--           LABEL.Labels = {Text}                                      Content node
--23--           FONT.Name = Helvetica  FONT.Points = 10                     containing
--24--           FONT.BaselineOffset = 0                                     36 characters
--25--           FONT.Boldness = Regular
--26--           FONT.Italic = false  FONT.Strikeout = false
--27--           FONT.Underlining = false
--28--           <Textportion1>}
--29--       {CHARS$  FONT$  LABEL$ ...                                     Content node
--30--           LABEL.Labels = {Text}                                       containing
--31--           FONT.Name = Helvetica  FONT.Points = 10                     10 characters
--32--           FONT.BaselineOffset = 0
--33--           FONT.Boldness = Regular
--34--           FONT.Italic = true  FONT.Strikeout = false
--35--           FONT.Underlining = true
--36--           <EmphasizedText>}
--37--       {CHARS$  FONT$  LABEL$ ...                                     Content node
--38--           LABEL.Labels = {Text}                                       containing
--39--           FONT.Name = Helvetica  FONT.Points = 10                     38 characters
--40--           FONT.BaselineOffset = 0
--41--           FONT.Boldness = Regular
--42--           FONT.Italic = false  FONT.Strikeout = false
--43--           FONT.Underlining = false
--44--           <Textportion2>}
--45--       {CHARS$ ... LABEL$  LABEL.Labels = {Footnote} ...              Content node
--46--           <Footnotetext>}}
```

† Notational remark: A pour process handles internalized script. Here and in the following examples internalized scripts are described in publication encoding. These descriptions will not be complete. Tags and bindings that are not relevant for the actual discussion are omitted; this will be indicated by an ellipsis (...).

First major step: Creating nodes

The pour operation creates boxes by traversing the template (lines 3 .. 20) in depth-first order. It selects a line box from the repetition node (line 3) and puts, as required (in line 5), the first character box into the line box. For reasons which lie beyond the scope of Interscript, it fills that line box with an additional 41 character boxes. (The CHARS node in line 14 is itself a template offering a repetition of character boxes.) It repeats this procedure creating a second line box with 42 character boxes. Then it stops creating boxes. Example 8.2 shows the result.

Example 8.2: Node structure created according to the template in Example 8.1

```
--1--   {BOX$ LABEL$ ...                                              First line box
--2--      LABEL.LabelSet = {TextLine}
--3--      {CHAR$ FONT$ MOLD$ LABEL$ ...                              First character box
--4--            LABEL.Labels = {Text}
--5--            FONT.Name = Roman  FONT.Points = 12
--6--            FONT.BaselineOffset = 0
--7--            FONT.Boldness = MATCH
--8--            FONT.Italic = MATCH  FONT.Strikeout = MATCH
--9--            FONT.Underlining = MATCH
--10--           MOLD.Coercion = "FONT.Boldness = FONT.Boldness# + 1"}
--11--     {CHAR$ FONT$ MOLD$ LABEL$ ...                              Following
--12--           LABEL.Labels = {Text}                                 character boxes
--13--           FONT.Name = Roman  FONT.Points = 12                   (repeated 41 times)
--14--           FONT.BaselineOffset = 0
--15--           FONT.Boldness = MATCH
--16--           FONT.Italic = MATCH  FONT.Strikeout = MATCH
--17--           FONT.Underlining = MATCH }
--18--     ....}
--19--   {BOX$ LABEL$ ...                                              Second line box
--20--      LABEL.LabelSet = {TextLine} ...}                           Same as the first one
```

Of course, this is not the only legal construction. Creating 84 line boxes containing one character box each would also be legal.

Second major step: Filling nodes

The pour operation performs filling in six steps:

Step 1: There is only one relevant label: Text. The pour operation builds one MOLD node stream consisting of the 42 character boxes of the first line followed by the 42 character boxes of the second line.

Step 2: The content node stream attached to the relevant label Text consists of the 84 character boxes of Textportion1, EmphasizedText, and Textportion2.

Step 3: The content node stream matches the MOLD node stream. The content nodes forming the Footnotetext do not carry any relevant label. They are not handled by this pour and are appended to the resulting two line boxes.

Step 4: The content character boxes carry all the tags of the corresponding MOLD character boxes (except the MOLD$ tag).

Step 5: The content character boxes replace the corresponding MOLD character boxes. The bindings of FONT.Name and FONT.Points are changed to the corresponding values in the MOLD character boxes.

Step 6: The first character box in each line contains a MOLD.Coercion expression. The evaluation changes the binding of FONT.Boldness from its actual value Regular into the new value Darker.

Example 8.3: Result of the pour operation of Example 8.1.

```
--1--   {BOX$ LABEL$ ... LABEL.LabelSet = {TextLine}          First line box
--2--      {CHAR$ FONT$ ...                                   First character box
--3--          FONT.Name = Roman  FONT.Points = 12             of Textportion1
--4--          FONT.BaselineOffset = 0
--5--          FONT.Boldness = Darker
--6--          FONT.Italic = false  FONT.Strikeout = false
--7--          FONT.Underlining = false
--8--          <..>}
--9--       {CHARS$ FONT$ ...                                 This is a sequence of
--10--          FONT.Name = Roman  FONT.Points = 12            35 character boxes
--11--          FONT.BaselineOffset = 0                        of Textportion1
--12--          FONT.Boldness = Regular
--13--          FONT.Italic = false  FONT.Strikeout = false
--14--          FONT.Underlining = false
--15--          <..>} .....
--16--       {CHARS$ FONT$ ...                                 This is a sequence of
--17--          FONT.Name = Roman  FONT.Points = 12            6 character boxes
--18--          FONT.BaselineOffset = 0                        of EmphasizedText
--19--          FONT.Boldness = Regular
--20--          FONT.Italic = true  FONT.Strikeout = false
--21--          FONT.Underlining = true
--22--          <..>} ..... }
--23--   {BOX$ LABEL$ ... LABEL.Labels = {Textline}            Second line box
--24--      {CHAR$ FONT$ ...                                   Seventh character box
--25--          FONT.Name = Roman  FONT.Points = 12             of EmphasizedText
--26--          FONT.BaselineOffset = 0
--27--          FONT.Boldness = Darker
--28--          FONT.Italic = false  FONT.Strikeout = false
--29--          FONT.Underlining = false
--30--          <..>}
--31--       {CHARS$ FONT$ ...                                 This is a sequence of
--32--          FONT.Name = Roman  FONT.Points = 12            3 character boxes
--33--          FONT.BaselineOffset = 0                        of EmphasizedText
--34--          FONT.Boldness = Regular
--35--          FONT.Italic = true  FONT.Strikeout = false
--36--          FONT.Underlining = true
--37--          <..>}
--38--       {CHARS$ FONT$ ...                                 This is a sequence of
--39--          FONT.Name = Roman  FONT.Points = 12            38 character boxes
--40--          FONT.BaselineOffset = 0                        of Textportion2
--41--          FONT.Boldness = Regular
--42--          FONT.Italic = false  FONT.Strikeout = false
--43--          FONT.Underlining = false
--44--          <..>} ..... }
--45--   {CHARS$ LABEL$ ... LABEL.Labels = {Footnote} ...      Left-over
--46--          <Footnotetext>}
```

The result, as shown in Example 8.3, is a sequence of two line boxes with the label TextLine, and a left-over node labelled Footnote. Higher pour operations may handle these sequence elements according to their label.

8.2.5 An example of a multi-level pour

This section illustrates how streams are formed in multi-level POUR nodes. Figures 8.2 and 8.3 show a two-level pour producing "chapters with marginal notes." These chapters consist of a chapter title, a marginal note title, and several paragraphs containing a text part and a marginal note part.

The inner POUR nodes handle paragraphs and produce lines for the text part (their POUR label is indicated by a black frame) and for the marginal note part (gray frame). The outer POUR node pours these lines together with the chapter title (dark gray frame) and marginal note title (light gray frame) into pages. Figure 8.2 shows the structure of the outer POUR node after the inner pours are done.

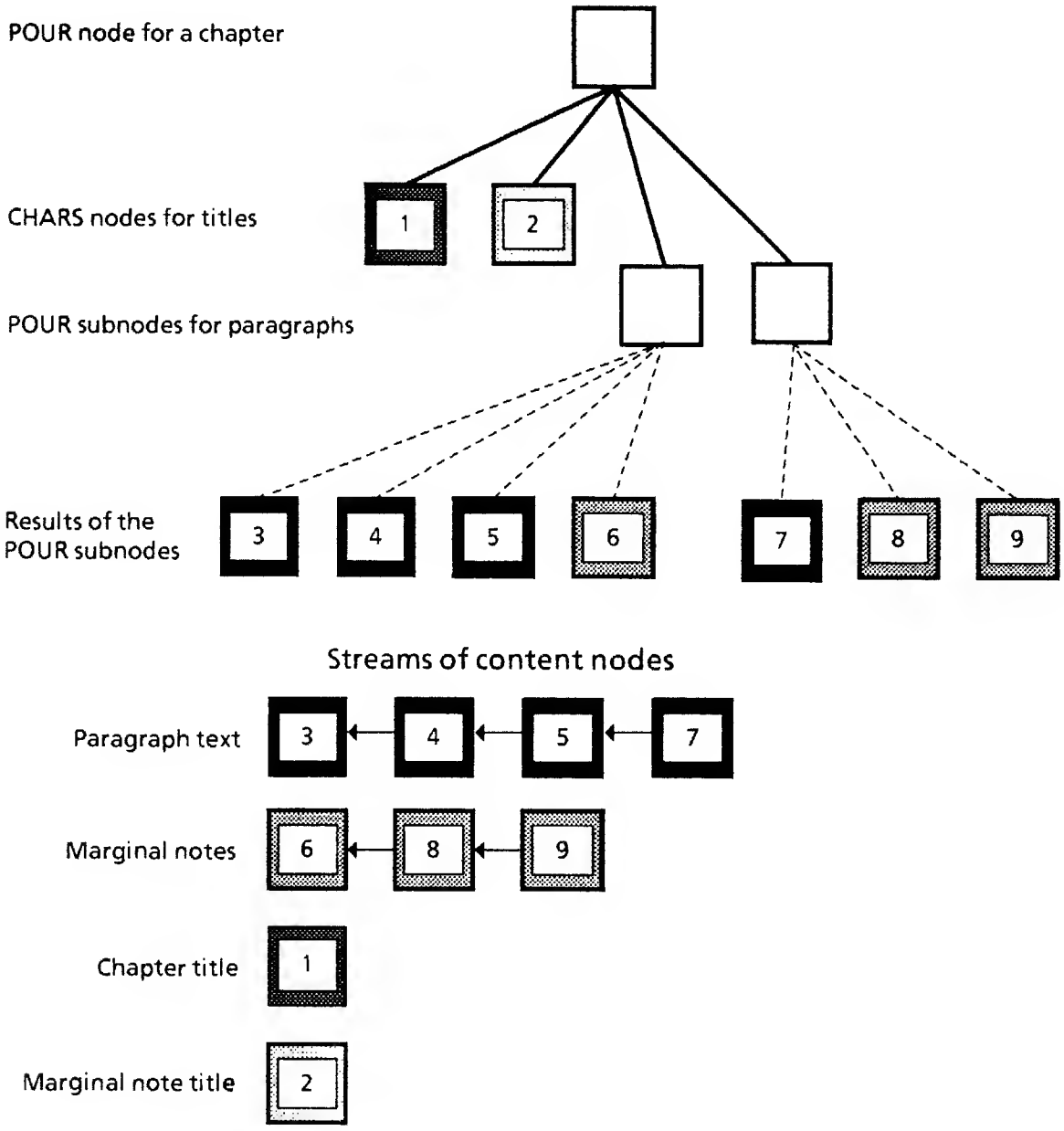

Figure 8.2: Streams in a POUR node

The chapters of Figure 8.2 are laid out into pages with two columns, one for text and one for marginal notes. The first page has special column lines for the titles. Figure 8.3a shows a page box sequence before the content nodes are poured in and Figure 8.3b shows the result of the pour.

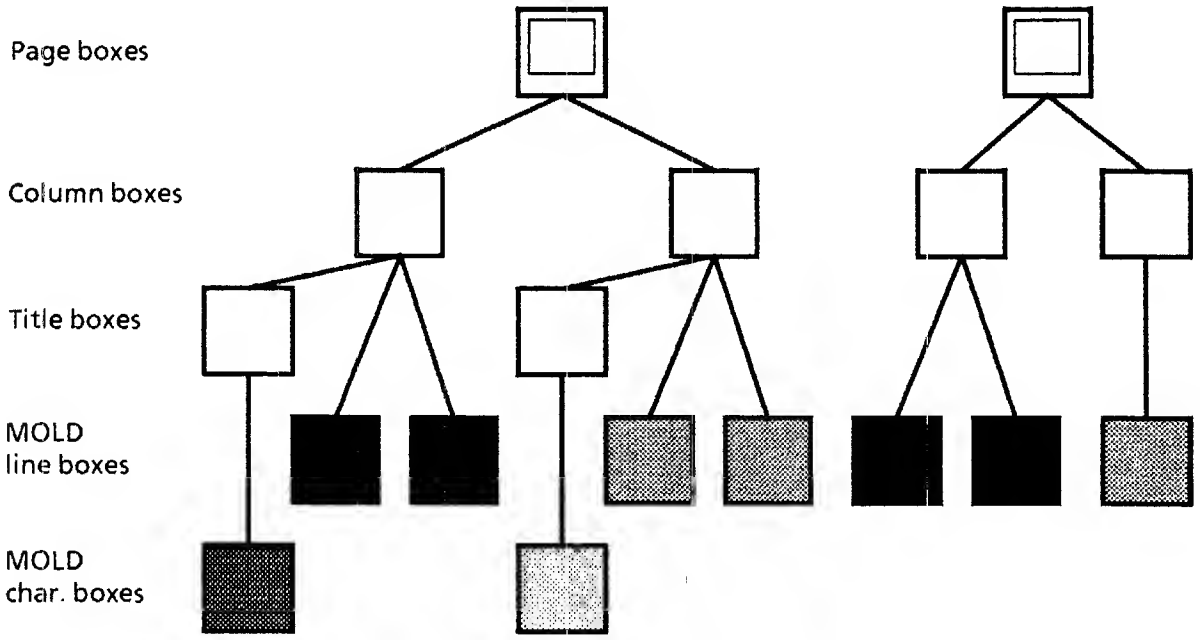

Figure 8.3a: Layout boxes before pouring

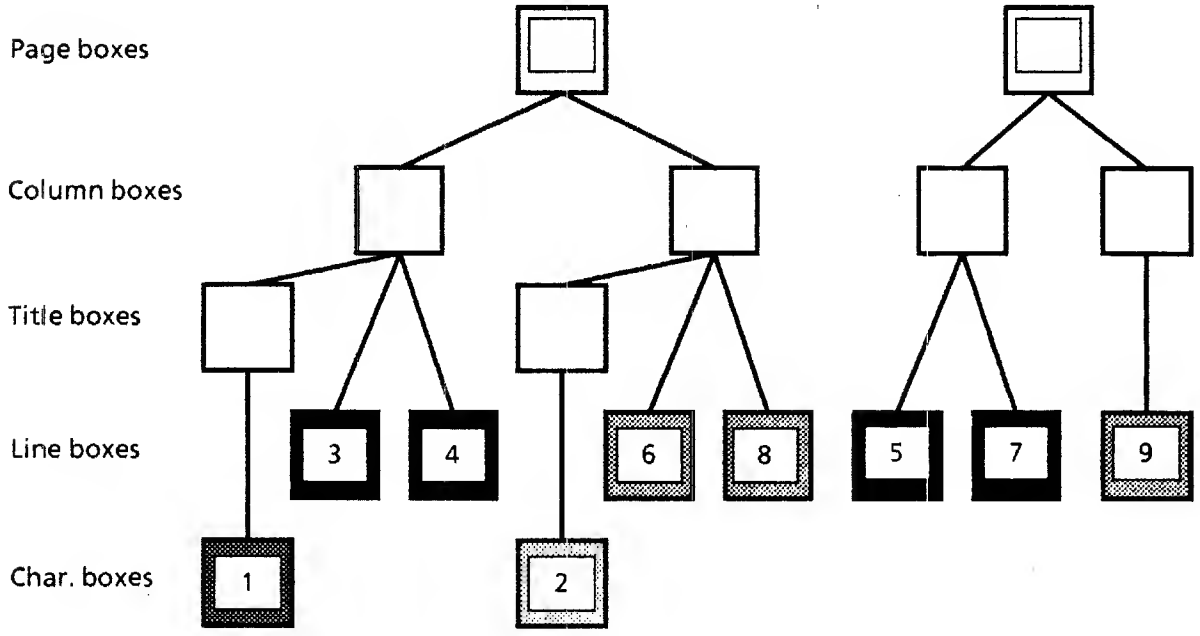

Figure 8.3b: Layout boxes after pouring

8.3 Penalties

Penalties control layout decisions. Penalties may come from two sources:

- Penalties may be explicitly attached to nodes by the PENALTY$ tag. Normally they are attached to choice and repetition nodes in templates. They may express a bias for certain alternatives and fewer repetitions.
- Penalties may be generated by TOGETHER nodes contained in POUR nodes. TOGETHER nodes expect their subnodes to be rendered as "near" as possible.

A formatting process is expected to produce an image format with a "minimal" penalty sum.

8.3.1 Calculating penalty sums

When the formatting process has completed the pour process, it calculates the associated penalty sum either for the image format as a whole or separately for each part as framed by fences.

Penalties given by PENALTY$ tags are simply added together.

Penalties generated byTOGETHER nodes are calculated by this procedure:

> For each immediate subnode of a TOGETHER node the formatting process considers the box in the image format that the subnode was poured into. Such a box is called an *associated box*. For each *pair of associated boxes* the pour operation calculates a layout distance. The LevelLabel attribute of the TOGETHER$ tag determines which calculation method to use.
>
> If the LevelLabel attribute is bound to NIL, the layout distance is 0 if the parent boxes of the box pair coincide; else the layout distance is 1.
>
> If the LevelLabel attribute is bound to an atom, the layout distance counts the number of boxes "between" the box pair. Not all boxes in the image format contribute to this distance; the relevant ones are labelled with the atom given by the LevelLabel attribute. More precisely, the pour operation traverses the image format in depth-first order and counts the number of relevant box nodes it encounters between the box pair. The layout distance is either zero (if there is no relevant box or if the parent boxes of the box pair coincide) or one less than the number of relevant boxes between the box pair.
>
> The layout distance is then multiplied by the number given by the Penalty attribute of the TOGETHER$ tag. This procedure is performed for each pair of associated boxes and the resulting numbers are added together. This sum is added to the penalty sum of the solid layout.

8.3.2 Minimizing penalty sums

To find an image format with a "minimal" penalty sum, a formatting process must start several pour processes. In principle, it must exhaust all the possibilities offered by templates before deciding which image format to choose. Since the number of legal image formats is typically very large, Interscript provides a selection algorithm that supplies a local minimum for the penalty sum instead of a global one. An editor is deemed successful with any image format whose penalty sum is at or below a local minimum. Of course, an editor may search further for better image formats. So even with penalties a script does *not uniquely specify* the appearance of its content; that depends on the cleverness of an editor.

Minimizing penalties for a single selection

The principles of the selection algorithm can be demonstrated in the simple case where only a single selection is to be optimized.

All the alternatives of a single selection are regarded as *threaded in a sequence*. The alternatives of a *choice* are already threaded and a *repetition* gives the infinite sequence of the alternatives {no repetition}, {one repetition}, {two repetitions}, etc. The search for a good alternative always starts by examining the first alternative and continues to the next in sequence. For each alternative the selection algorithm calculates the penalty.

The selection algorithm stops examining the sequence of alternatives when

- it finds an alternative with a penalty less than SatisfactionThreshold (defined by the actual POUR node)

AND

- EITHER the next n alternatives don't have a better penalty (n is given by the SatisfactionForwardSearch attribute of the actual POUR node)
- OR one of the next alternatives is at least d worse in penalty (d is given by the SatisfactionUpwardSearch attribute of the actual POUR node).

Selections that cause an unsuccessful pour are considered "infinitely" bad (i.e. they get the highest possible penalty).

Minimizing penalty sums for image formats or fenced parts of image formats

When constructing an image format, selections may occur on all levels. The decision as to which alternative to choose can only be made when considering the entire image format or, if fences break the layout into parts, an entire part. An alternative, which is chosen in a subordinate selection and is "good" for that level, may prove unsatisfactory at a higher level.

The alternatives that occur when constructing an image format are considered a *sequence* threaded in depth-first order. Alternatives occuring on lower levels must be propagated to the whole. In this way a sequence of multi-level alternatives is constructed with associated penalty sums. To this sequence the above approximation procedure is applied (the necessary values for SatisfactionThreshold, SatisfactionForwardSearch, and SatisfactionUpwardSearch are supplied by some global POUR node).

A precise description of the selection algorithm is given in the Standard.

8.4 Fixing process

Fixing is the second major step of the formatting process; it handles the incomplete image formats that the pour process produced. It first calculates synthesized box measures and then fixes the positions and sizes of all the boxes. The complete image format is the result of this step.

Calculating synthesized measures

For each box with synthesized measures the numeric procedure (described in section 8.5.3) calculates numeric measures that replace the synthesized ones. The numeric measures are arranged in such a way that the subboxes contained in a synthesized box just fit. The numeric procedure starts with the leaf boxes in the image format and continues upward.

Fixing measures

The fixing procedure (described in section 8.5.4) fixes the position and the size of the boxes. Not all 36 measure values associated with a box are needed to describe the result. Six values are sufficient: three for the x-direction and three for the y-direction. They are:

- the Nominal values of the LowPartExtent measure
- the Nominal values of the HighPartExtent measure
- the Nominal values of the FromLowerContainer measure.

The fixing procedure calculates these nominal values by squeezing or stretching the original nominal values. The fixing operation fails when the calculated values exceed shrinkability. Then the formatting process must start another pour process trying other alternatives.

The fixing procedure starts with PAGE boxes in the image format and traverses the internal tree of subboxes in depth-first order. It is a straightforward process; having fixed the measures of a box, it proceeds to the subordinate boxes and fixes their measures. PAGE boxes as starting points always have fixed measures.

The following sections contain narrative descriptions of some technical procedures used by the fixing process.

8.4.1 The compromise function

The compromise function is used when the nominal values of some measures are required to fulfill a certain equation that the actual values don't fulfill. The compromise function calculates new nominal values that fulfill the desired equation.

A typical problem is the *glue* between two consecutive boxes that are laid out in the up direction and have contradictory sibling distances.

Example 8.4: Contradictory sibling distances.

```
--1--   {BOX$ BOX.XSpan = {SPAN$ ... SPAN.FromLowSibling = {MEASURE$ 0 60 100} ...}
--2--   {BOX$ BOX.XSpan = {SPAN$ ... SPAN.FromHighSibling = {MEASURE$ 0 50 90} ...}
```

The sibling distances are to be replaced by a common distance (the glue), but what should this distance be? The Nominal value of the first box requires 60, that of the second box 50.

The compromise function calculates a new Nominal value as a "fair" compromise between the original distances by squeezing the larger one and stretching the smaller one. It takes into account how shrinkable (given by the difference between Nominal value and Under value) and stretchable (given by the difference between Over value and Nominal value) the nominal values are. The greater the shrinkability and the stretchability are, the more the Nominal value will be changed. This is the reason why Over values are important, although they do not provide an upper bound for stretching. In the above example the compromise function yields the Nominal value 54.

The compromise function will not change fixed measures. On the other hand, if there is an arbitrarily stretchable measure involved in a compromise and the changeability of the other measures is much less, only that arbitrarily stretchable measure will be changed.

Measures may arbitrarily be stretched but they cannot shrink to less than their Under value. Therefore a compromise is sometimes not possible. If, in the above example, the distance measure of the higher box were fixed (say {MEASURE$ 50}) and the Under value of the lower box were greater than this fixed distance (say {MEASURE$ 100 200 300}), a compromise within the permitted scope would not be possible.

The technical description of the compromise function is given in the Standard.

8.4.2 The glue procedure

The glue procedure is used when boxes are filled according to the INSIDELAYOUTMETHOD up or down. It glues two consecutive boxes together by calculating a glue measure that defines the distance between the two boxes. This glue measure replaces the corresponding sibling distance measures.

The glue measure is calculated from:

- for the up method: the higher sibling distance of the first box and the lower sibling distance of the second box.
- for the down method: the lower sibling distance of the first box and the higher sibling distance of the second box.

Glue measures for fixed sibling distances

If one of the sibling distances is fixed, it is taken as a glue measure. If both sibling distances are fixed and different, a fixing error occurs.

Glue measures for stretchable sibling distances

If both sibling distances are variable, the glue measure is variable too. It is defined as:

- the Under value
  It is the maximum of the Under values of the sibling distances.
- the Nominal value
  The SiblingAdjacency attribute of the INSIDELAYOUT$ tag determines what compromise function must be chosen to get the Nominal value. If this attribute is bound to SERIAL, the Nominal value is half the sum of the two original Nominal values. If this attribute is bound to PARALLEL, the compromise function of section 8.5.1 is used.
- the Over value
  It is the maximum of the Over values of the sibling distances.

The compromise function of section 8.5.1, when used to calculate the glue measure, guarantees that the Nominal value of the glue measure is larger than the Under value and smaller than the Over value. When the arithmetic mean is used, it may happen that the resulting measure triple does not satisfy these inequalities and that causes a pour error.

The following examples explain the glue procedure. Example 8.5 shows two subboxes to be glued together in a "line" box . Only the sibling distances in x-direction (the higher one of the first subbox and the lower one of the second) are relevant.

To simplify notation, we use these abbreviations:

LPE	LowPartExtent	HPE	HighPartExtent
FLS	FromLowSibling	FHS	FromHighSibling
FLC	FromLowContainer	FHC	FromHighContainer

Example 8.5: Boxes in a line before being glued.

```
--1--   {BOX$ INSIDELAYOUT$ ...
--2--      INSIDELAYOUT.X = {INSIDELAYOUTMETHOD.Direction = up
--3--                        INSIDELAYOUTMETHOD.SiblingAdjacency = Serial}
--4--      {BOX$ ...
--5--         BOX.X = {SPAN$ SPAN.FLS = {MEASURE$ 0 254 508}
--6--                        SPAN.FHS = {MEASURE$ 0 254 508} ...} ...}
--7--      {BOX$ ...
--8--         BOX.X = {SPAN$ SPAN.FLS = {MEASURE$ 0}
--9--                        SPAN.FHS = {MEASURE$ 200 400 600}...} ...}}
```

Example 8.6 shows the result of the glue procedure.

Example 8.6: The boxes of Example 8.5 after being glued.

```
--1--   {BOX$ INSIDELAYOUT$ ...
--2--      INSIDELAYOUT.X = {INSIDELAYOUTMETHOD.Direction = up
--3--                        INSIDELAYOUTMETHOD.SiblingAdjacency = Serial}
--4--      {BOX$ ...
--5--         BOX.X = {SPAN$ SPAN.FLS = {MEASURE$ 0 254 508}
--6--                        SPAN.FHS = {MEASURE$ 200 327 600} ...} ...}
--7--      {BOX$ ...
--8--         BOX.X = {SPAN$ SPAN.FLS = {MEASURE$ 200 327 600}
--9--                        SPAN.FHS = {MEASURE$ 0} ...} ...}}
```

8.4.3 The numeric procedure

The size measures of a box (i.e. LowPartExtent and HighPartExtent in the x- and y-direction) may be synthesized. The numeric procedure replaces these synthesized measures by numeric ones. Such a numeric measure depends on the poured-in subboxes and the direction attribute of the INSIDELAYOUT$ tag. It is calculated such that the sizes and distances of the subboxes need not shrink nor stretch.

1. Boxes carrying an INSIDELAYOUT$ tag with the method fixed must not have synthesized measures in this version of Interscript.
2. Boxes carrying an INSIDELAYOUT$ tag with the method onOrigins require that a "low sum" and a "high sum" be calculated *for each subbox*. The "low sum" adds the LowPartExtent and the FromLowContainer measure. The "high sum" adds the HighPartExtent and the FromHighContainer measure. The LowPartExtent (or HighPartExtent) measure of the containing box is then defined by these rules:
   - the Under value
     The maximum under value appearing in the "low sums" (or "high sums") of the subboxes is taken.
   - the Nominal value
     The maximum nominal value appearing in the "low sums" (or "high sums") of the subboxes is taken.
   - the Over value
     The maximum over value appearing in the "low sums" (or "high sums") of the subboxes is taken.

   When a box shrinks, the measures of its subboxes must also shrink. The above definition guarantees that subboxes can shrink without violating their shrinkability as long as the containing box remains within its shrinkability scope.

3. Boxes carrying an INSIDELAYOUT$ tag with the method up require that the subboxes be glued together by the glue procedure of section 8.5.2.
   - The LowPartExtent measure of the containing box is defined as the sum of the LowPartExtent and the FromLowContainer measure of the first subbox.
   - The HighPartExtent measure of the containing box is also defined as a sum that adds the HighPartExtent measure of the first subbox, the LowPartExtent, the HighPartExtent, and the *glue* measure for all the following subboxes, and the FromHighContainer measure of the last subbox.
4. For boxes carrying an INSIDELAYOUT$ tag with the method down the numeric procedure is the same as for the up method with "low" and "high" interchanged.

The following examples explain the numeric procedure. Example 8.7 shows a "column" box with synthesized height containing two line boxes. Only the measures in y-direction are relevant.

Example 8.7: Column box with synthesized height

```
--1--   {BOX$ INSIDELAYOUT$ ...
--2--       INSIDELAYOUT.Y = {INSIDELAYOUTMETHOD.Direction = down
--3--                         INSIDELAYOUTMETHOD.SiblingAdjacency = Serial}
--4--       BOX.Y = {SPAN$  SPAN.LPE = SYNTHESIZED
--5--                         SPAN.HPE = SYNTHESIZED ...}
--6--       {BOX$ ...
--7--           BOX.Y = {SPAN$ SPAN.FHC = {MEASURE$ 0}
--8--                         SPAN.FLS = {MEASURE$ 0 0 508}
--9--                         SPAN.LPE = {MEASURE$ 254}
--10--                        SPAN.HPE = {MEASURE$ 508} ...} ...}
--11--      {BOX$ ...
--12--          BOX.Y = {SPAN$ SPAN.FLC = {MEASURE$ 0}
--13--                        SPAN.FHS = {MEASURE$ 0 254 508}
--14--                        SPAN.LPE = {MEASURE$ 254}
--15--                        SPAN.HPE = {MEASURE$ 508} ...} ...}}
```

After having glued together the two line boxes with the glue measure {MEASURE$ 0 127 508}, the numeric procedure calculates the synthesized height. Example 8.8 shows the result.

Example 8.8: The column box of Example 8.7 with numeric height

```
--1--   {BOX$ INSIDELAYOUT$ ...
--2--      INSIDELAYOUT.Y = {INSIDELAYOUTMETHOD.Direction = down
--3--                        INSIDELAYOUTMETHOD.SiblingAdjacency = Serial}
--4--      BOX.Y = {SPAN$  SPAN.LPE = {MEASURE$ 1216 1343 1724}
--5--                        SPAN.HPE = {MEASURE$ 508} ...}
--6--      {BOX$ ...
--7--         BOX.Y = {SPAN$ SPAN.FHC = {MEASURE$ 0}
--8--                        SPAN.FLS = {MEASURE$ 0 127 508}
--9--                        SPAN.LPE = {MEASURE$ 254}
--10--                       SPAN.HPE = {MEASURE$ 508} ...} ...}
--11--     {BOX$ ...
--12--        BOX.Y = {SPAN$ SPAN.FLC = {MEASURE$ 0}
--13--                       SPAN.FHS = {MEASURE$ 0 127 508}
--14--                       SPAN.LPE = {MEASURE$ 254}
--15--                       SPAN.HPE = {MEASURE$ 508} ...} ...}}
```

8.4.4 The fixing procedure

The fixing procedure is performed independently for the x- and y-direction. It traverses the image format in depth-first order replacing box measures by a triple of numbers for the x-direction and a triple for the y-direction. Each triple consists of the Nominal values LowPartExtent, the HighPartExtent, and the FromLowContainer measure.

The fixing procedure proceeds in top-down fashion. In principle, it takes the fixed extent measures of a containing box and changes the Nominal values of some subbox measures such that certain equations are fulfilled. Which subbox measures must be chosen and which equations must be fulfilled depends on the direction attribute of the INSIDELAYOUT$ tag of the containing box.

The fixing procedure uses the compromise function

- to stretch the original Nominal values, if they are too small. A solution within the scope of stretchability is always possible.
- to squeeze the original Nominal values, if they are too large. A solution within the scope of shrinkability is only possible if the corresponding sum of the under values does not exceed the fixed values of the containing box.

To simplify notation, we abbreviate a Nominal value as ".N".

1. When the INSIDELAYOUT$ tag has the method fixed, each subbox is laid out as if it were the only subbox in the containing box.

   The nominal values of the FHC, FLC, LPE, and HPE measure of a subbox must be changed to fulfill the following equation:

   FHC.N + FLC.N + LPE.N + HPE.N = fixed width of the containing box

2. When the INSIDELAYOUT$ tag has the method onOrigins, each subbox is laid out as if were the only subbox in the containing box.

   The Nominal values of the FLC and LPE measure of a subbox must be changed to fulfill the following equation:

   FLC.N + LPE.N = fixed LPE.N of the containing box

In the same way the Nominal values of the HLC and HPE measure of a subbox must be changed to fulfill the following equation:

FHC.N + HPE.N = fixed HPE.N of the containing box.

3. When the INSIDELAYOUT$ tag has the method up, the subboxes cannot be laid out separately. First they are glued together by the glue procedure. The fixing procedure then examines the Nominal values of the following measures:

   - the FLC measure of the first subbox;
   - the LPE, HPE, and the glue measure of all subboxes (each glue measure counts only once);
   - the FHC measure of the last subbox.

   The fixing procedure changes these Nominal values such that their sum becomes equal to the fixed width (or height) of the containing box. After that, the extent measures and the position of each subbox are uniquely determined.

   Then the fixing procedure calculates the FLC distance of each subbox. That is already done for the first subbox. The FLC distance for the following subboxes are recursively calculated by adding the FLC distance of the proceeding subbox and the Nominal value of the glue measure.

   Note, calculating the FLC distances of the second and the following subboxes is a straightforward addition and *does not change positions*. The fixing procedure does NOT check if those FLC distances violate their original stretchability scope.

4. When the INSIDELAYOUT$ tag has the method down, the fixing procedure is the same as the up method with "low" and "high" interchanged.

The following examples explain the fixing procedure. Example 8.9 shows a fixed "column" box containing two line boxes. Only the measures in the y-direction are relevant.

Example 8.9: Fixed column box with stretchable line boxes

```
--1--   {BOX$ INSIDELAYOUT$ ...
--2--      INSIDELAYOUT.Y = {INSIDELAYOUTMETHOD.Direction = down
--3--                        INSIDELAYOUTMETHOD.SiblingAdjacency = Serial}
--4--      BOX.Y = {SPAN$  SPAN.LPE = {MEASURE$ 1524}
--5--                      SPAN.HPE = {MEASURE$ 508} ...}
--6--      {BOX$ ...
--7--         BOX.Y = {SPAN$ SPAN.FHC = {MEASURE$ 0}
--8--                        SPAN.FLC = {MEASURE$ 0}
--9--                        SPAN.FLS = {MEASURE$ 0 0 S08}
--10--                       SPAN.LPE = {MEASURE$ 254}
--11--                       SPAN.HPE = {MEASURE$ 508} ...} ...}
--12--     {BOX$ ...
--13--        BOX.Y = {SPAN$ SPAN.FHC = {MEASURE$ 0}
--14--                       SPAN.FLC = {MEASURE$ 0}
--15--                       SPAN.FHS = {MEASURE$ 0 0 S08}
--16--                       SPAN.LPE = {MEASURE$ 254}
--17--                       SPAN.HPE = {MEASURE$ 508} ...} ...}}
```

First, the subboxes are glued together by the glue measure {MEASURE$ 0 127 S08}. Then the fixing procedure changes the FHC, LPE, and HPE measures of the first subbox, the glue measure, and the FLC, LPE, and HPE measures of the second subbox, such that the sum of

their Nominal values is equal to the fixed height {MEASURE$ 2032} of the outer box. Since the glue measure is the only changeable measure, it is stretched to {MEASURE$ 508}. Finally the fixing procedure calculates the FLC distance of the first subbox as {MEASURE$ 1270}. Note that the original FLC distance {MEASURE$ 0} of the first subbox is not relevant here. Example 8.10 shows the result.

Example 8.10: The column box of Example 8.9 with fixed line boxes

```
--1--   {BOX$ INSIDELAYOUT$ ...
--2--       INSIDELAYOUT.Y = {INSIDELAYOUTMETHOD.Direction = down
--3--                         INSIDELAYOUTMETHOD.SiblingAdjacency = Serial}
--4--       BOX.Y = {SPAN$  SPAN.LPE = {MEASURE$ 1524}
--5--                         SPAN.HPE = {MEASURE$ 508} ...}
--6--       {BOX$ ...
--7--           BOX.Y = {SPAN$ SPAN.FLC = {MEASURE$ 1270}
--8--                         SPAN.LPE = {MEASURE$ 254}
--9--                         SPAN.HPE = {MEASURE$ 508} } ...}
--10--      {BOX$ ...
--11--          BOX.Y = {SPAN$ SPAN.FLC = {MEASURE$ 0}
--12--                        SPAN.LPE = {MEASURE$ 254}
--13--                        SPAN.HPE = {MEASURE$ 508} } ...}}
```

8.5 Another example

This section *outlines* how an existing editor does pouring. It demonstrates the major steps by means of a simple document consisting of a title and two text sections. This document will be rendered into one-column pages, the first of which contains a fixed heading box for the title.

Example 8.11 shows this document expressed in Interscript. To demonstrate nested pours, it uses a two level pour, the outer one producing pages and the inner one lines. (This simple document could also be rendered by a one-level pour.) Again the notational conventions explained in the footnote of Example 8.1 are observed.

Example 8.11: A simple document

```
--1--   {DOCUMENT$ DOCUMENT.Title = {CHAR$ ... <Pour example>} ...
--2--     {POUR$ POUR.Labelset = {Title Line}                    Outer POUR node
--3--       POUR.template =
--4--       {TEMPLATE$ TEMPLATE.Expresses = sequence
--5--         {BOX$ PAGE$ TEMPLATE$ ...                         Special first page
--6--           TEMPLATE.Expresses = sequence ...
--7--           {BOX$ ...                                       Page heading box
--8--               -- some fixed box measures --
--9--             {CHARS$ FONT$ MOLD$ LABEL$
--10--              LABEL.Labelset = {Title}... }}
--11--            {TEMPLATE$ TEMPLATE.Expresses = repetition ...  Repetition of lines
--12--              {BOX$ MOLD$ LABEL$ ...
--13--                 -- all box measures are bound to MATCH --
--14--               LABEL.Labelset = {Line}... }}
--15--        {TEMPLATE$ TEMPLATE.Expresses = repetition          Repetition of pages
--16--          {BOX$ PAGE$ TEMPLATE$ ...
--17--            TEMPLATE.Expresses = repetition ...            Repetition of lines
--18--            {BOX$ MOLD$ LABEL$ ...
--19--               -- all box measures are bound to MATCH --
--20--             LABEL.Labelset = {Line}... }}}}
--21--      {CHARS$ FONT$ LABEL$ LABEL.Labelset = {Title} ...    Document title
--22--        <a document title>}
--23--      {POUR$ POUR.Labelset = {Text}                        First inner POUR node
--25--        POUR.template =
--25--        {TEMPLATE$ TEMPLATE.Expresses = repetition
--26--            {BOX$ LABEL$ ...                                 Lines
--27--               -- in x-direction it has fixed container distances
--28--                  and a zero LPE --
--29--               -- in y-direction it has fixed extent measures --
--30--             LABEL.Labelset = {Line}
--31--             {CHARS$ FONT$ MOLD$ LABEL$
--32--               LABEL.Labelset = {Text}... }}}
--33--        {CHARS$ FONT$ LABEL$                               Sequence of
--34--            LABEL.Labelset = {Text}...                      character boxes
--35--            <text of section one>}}}}
--36--      {POUR$  -- the same POUR node as above with          Second inner POUR node
--37--                 another textportion -- }}}
```

The editor does pouring in a top-down fashion. It creates boxes and calculates their measures as early as possible *minimizing*, but not excluding, the risk of later pour errors.

To construct the image format, the editor first opens a node and copies the DOCUMENT$ tag with its relevant bindings into that node. As it descends in the script, it meets the outer POUR node and activates a *pour operation*. This produces two pages containing the laid-out document. It puts those pages into the opened node and terminates pouring. Example 8.12 shows the result.

Example 8.12: Result of pouring the script in Example 8.11

```
--1--   {DOCUMENT$ DOCUMENT.Title = {CHAR$ ... <Pour example>} ...
--2--      {BOX$ PAGE$ -- some content -- }
--3--      {BOX$ PAGE$ -- some content -- } }
```

Next we discuss how the content of the page boxes is obtained.

Pour operation

For each POUR node a pour operation will be started during the pour process. Those pour operations act like coroutines. They may activate each other though only one is active at any moment. They preserve their present status when they return control; when activated again, they run as if they had not been interrupted.

Pour operations are invoked to supply nodes for the image format. Except for the highest nodes (normally page boxes) they are told to return nodes labelled with certain POUR labels. In principle, they rely on these five procedures:

- TEMPLATE **procedure**
  The template procedure is called to get the "next" node from the actual template. It traverses the template tree in depth-first order and returns the nodes it encounters together with information about the value of the Expresses attribute. On demand it terminates traversing choice and repetition nodes.

- MOLD **procedure**
  The MOLD procedure is called each time the TEMPLATE procedure supplies a MOLD node. It fills that MOLD node with the "next" content node with matching POUR labels that has not yet been poured. To get that content node, it traverses the actual POUR node in depth-first order examining the labelsets of all elementary content nodes and of *all* POUR subnodes (not just the highest POUR subnodes since there may be some "leftovers").

  An elementary content node is immediately poured without further action. When encountering a POUR subnode, the MOLD procedure activates the corresponding pour operation which creates and returns the desired content node. The MOLD procedure pours that content node into the MOLD node.

- VACUUM **procedure**
  The VACUUM procedure is called when the TEMPLATE procedure supplies a VACUUM node. It looks for VACUUMSOURCE nodes and pours them into the VACUUM node.

- BOX **procedure**
  The BOX procedure is called whenever a BOX node is to be put into the image format. As far as possible it calculates and fixes box measures checking if the actual box "fits" (i.e. does not violate the stretchability of the box measures involved). For example, it prevents character boxes from extending beyond a line (we assume FHC distances of character boxes are positive). The BOX procedure has access to all layout information available at the time it is invoked (e.g. it knows the page width when breaking lines).

- CHAR **procedure**
  The CHAR procedure is called to break running text into lines. From the BOX procedure it gets a sequence of character nodes fitting *geometrically* into a line. Its task is to produce *semantically* correct line breaks. It may activate a hyphenation procedure, add character boxes to the content (e.g. for a hyphenation sign), suppress other character boxes (e.g. blanks at line ends) etc. In particular, it may

choose other line breaks than those proposed by the BOX procedure causing the BOX procedure to do new calculations.

Here is how these procedures work together when producing an image format for the script in Example 8.8.

Pour operation for the outer POUR node

Step 1: The TEMPLATE procedure supplies the first page box.

Step 2: The TEMPLATE procedure, descending within the first page template, supplies the heading box which has fixed measures. The BOX procedure checks if the heading box fits into the page.

Step 3: The TEMPLATE procedure, descending within the heading box template, supplies the first character box which is a MOLD node with POUR label Title. The MOLD procedure scans the outer POUR node for matching nodes. The first one is a character box which is poured into the MOLD character box. The BOX procedure calculates the measures of the character box and states stretchabilty violations.

The third step is repeated until no character boxes are left (we assume the whole text fits into the heading box). The MOLD procedure then searches the inner POUR nodes. Since other matching nodes are not present, the pour operation terminates the filling of the heading box.

Step 4: The TEMPLATE procedure proceeding in the first page template, supplies a line box which is a MOLD node with POUR label Line. The MOLD procedure detects that label in the labelset of the first inner POUR node and starts a new pour operation, suspending the actual one.

Pour operation for the first inner POUR node

Step 1: The TEMPLATE procedure supplies a line of the inner POUR template. The BOX procedure calculates the line measures using the page width.

Step 2: The TEMPLATE procedure, descending within the line template, supplies a character box which is a MOLD node with POUR label Text. The MOLD procedure fills this MOLD node with a matching character box from the content. The BOX procedure calculates the measures of the character box and determines stretchability violations.

This procedure is repeated until the "line is filled" (i.e. the BOX procedure detects that even shrinking all the filled-in character boxes does not prevent the last character box from extending beyond the line). The CHAR procedure is called to fix the line break. The last character box is returned to the POUR node.

The pour operation returns the line and gets suspended.

Pour operation for the outer POUR node (continued)

Step 4 continues: The fourth step is repeated until the first inner POUR node cannot produce more lines (we assume the first section does not fill the first page). The MOLD procedure, looking for further content nodes with a POUR label of Line, activates the pour operation for the second inner POUR node, suspending the actual one.

Pour operation for the second inner POUR node

This pour operation produces lines in the same way as the first one. After returning a filled line, it gets suspended.

Pour operation for the outer POUR node (continued)

Step 4 continues: The fourth step is repeated until the page is filled (i.e. the BOX procedure detects that even shrinking all the filled-in lines does not prevent the last line from extending beyond the page). The last line is given back to the POUR node.

Step 5: The TEMPLATE procedure supplies the second page of the global template.

Step 6: Now the second page is filled with lines in the same way as the first page.

Appendix A

References

[1] Knuth, Donald E. *The TEXbook*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[2] Xerox Corporation. *Interpress Electronic Printing Standard*. Xerox System Integration Standard. Stamford, Connecticut; 1983 June; XSIS 048306.

[3] Xerox Corporation. *Interpress 82 Reader's Guide*. Xerox System Integration Guide. Stamford, Connecticut; 1982 May; XSIG 018205.

[4] Reid, Brian. *Scribe*. 7th Symposium on the Principles of Programming Languages. 1980

[5] Osanna, Joe F. *NROFF/TROFF User's Manual*. Computer Science Technical Report No. 54, Bell Laboratories.

[6] Xerox Corporation. *Character Code Standard*. Xerox System Integration Standard. Stamford, Connecticut; 1982 October; XSIS 058303.

[7] ECMA. *Office Document Architecture* (Standard 101). April 1985.

[8] CCITT. *Document Interchange Protocol for the Telematic Services* (Recommendation T.73). October 1984.